Overview of the Global Arrays Parallel Software Development Toolkit*:

Vinod Tipparaju

tipparajuv@ornl.gov

http://ft.ornl.gov/~vinod

Future Technologies and NCCS

Oak Ridge National Laboratory

* Contributors to these slides include Manoj Krishan (PNL), Bruce Palmer(PNL), and Jarek Nieplocha

Managed by UT-Battelle
for the Department of Energy

OAK RIDGE National Laboratory

Outline of the Tutorial

- Introduction Parallel programming models
- Global Arrays (GA) programming model
- GA Operations
  - Writing, compiling and running GA programs
  - Basic, intermediate, and advanced calls
    - With C and Fortran examples
- GA Hands-on session

People involved (not a complete list)

CRAY
THE SUPERCOMPUTER COMPANY

Ryan Olson
Kitrick Sheets

Late Dr. Jarek Nieplocha

Robert Harrison
harrisonrj@ornl.gov

IBM

Brian Smith
Mike Blocksome
Sameer Kumar

Manoj Krishnan
manoj@pnl.gov
(**leading GA effort at PNL)**

Bruce Palmer
bruce.palmer@pnl.gov

Abhinav Vishnu
abhinav.vishnu@pnl.gov

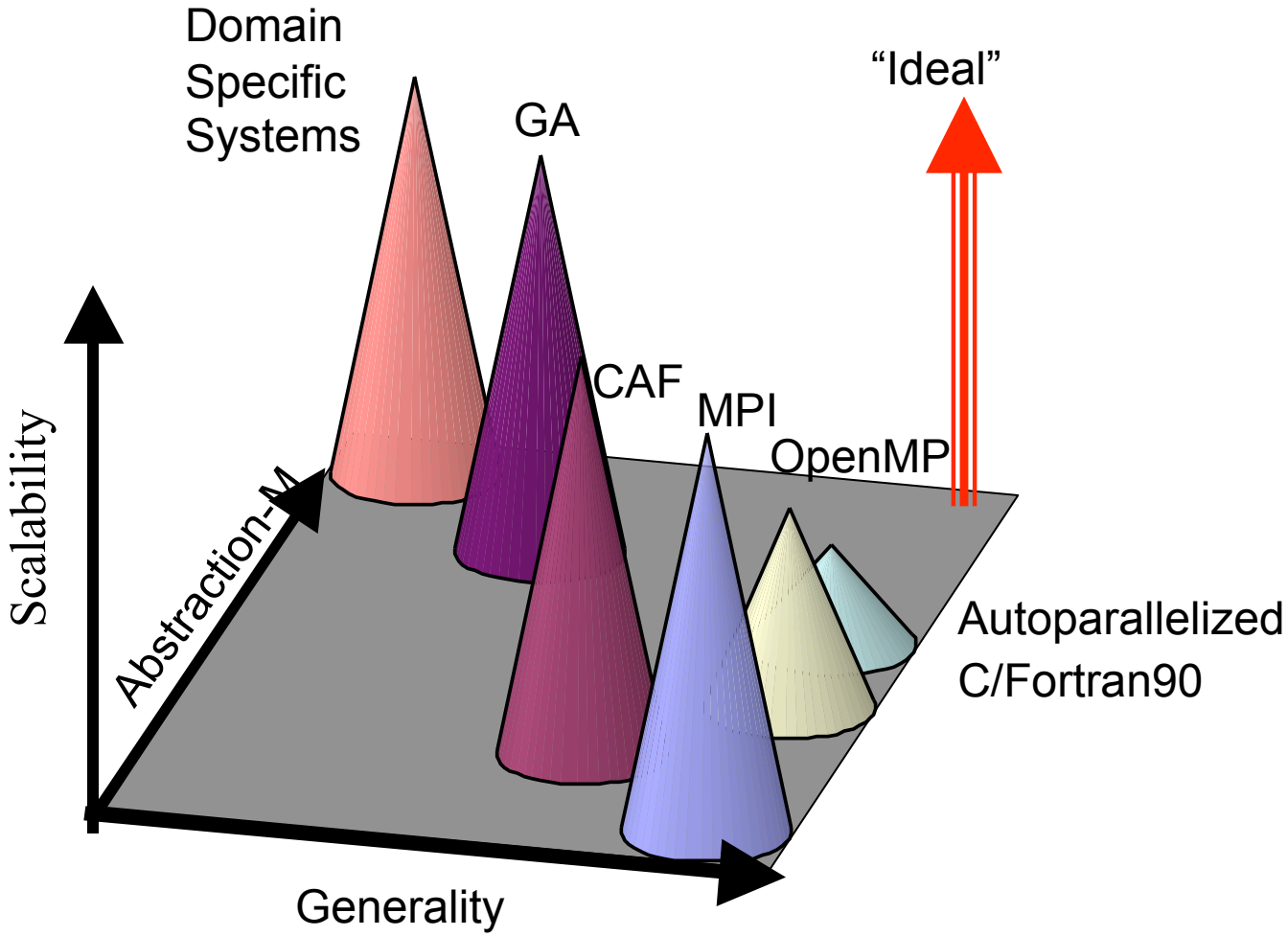
Edo Apra
Edo.apra@ornl.gov

Overview of the Global Arrays Parallel Software Development Toolkit: Introduction to Parallel Programming Models

Managed by UT-Battelle
for the Department of Energy

OAK RIDGE National Laboratory

Performance vs. Abstraction and Generality



Parallel Programming Models

- **Single Threaded**
  - **Data Parallel, e.g. HPF**
- **Multiple Processes**
  - **Partitioned-Local Data Access**
    - **MPI**
  - **Uniform-Global-Shared Data Access**
    - **OpenMP**
  - **Partitioned-Global-Shared Data Access**
    - **Co-Array Fortran**
  - **Uniform-Global-Shared + Partitioned Data Access**
    - **UPC, Global Arrays, X10**

High Performance Fortran

- **Single-threaded view of computation**
- **Data parallelism and parallel loops**
- **User-specified data distributions for arrays**
- **Compiler transforms HPF program to SPMD program**
  - **Communication optimization critical to performance**
- **Programmer may not be conscious of communication implications of parallel program**

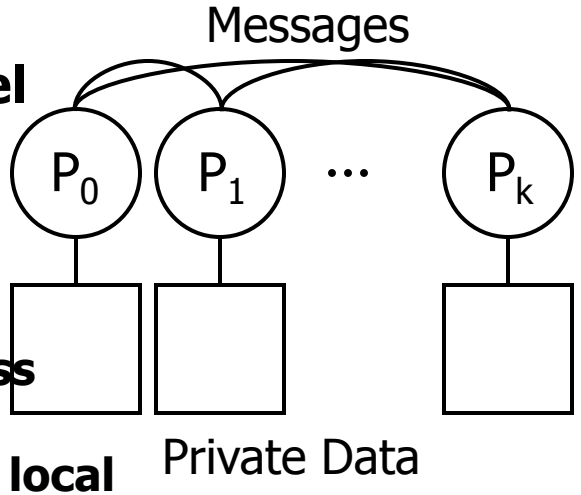
```
s=s+1
A(1:100) = B(0:99)+B(2:101)
HPF$ Independent
Do I = 1,100
   A(I) = B(I-1)+B(I+1)
End Do
```

```
HPF$ Independent
DO I = 1,N
HPF$ Independent
   DO J = 1,N
      A(I,J) = B(J,I)
   END
END
```

```
HPF$ Independent
DO I = 1,N
HPF$ Independent
   DO J = 1,N
      A(I,J) = B(I,J)
   END
END
```

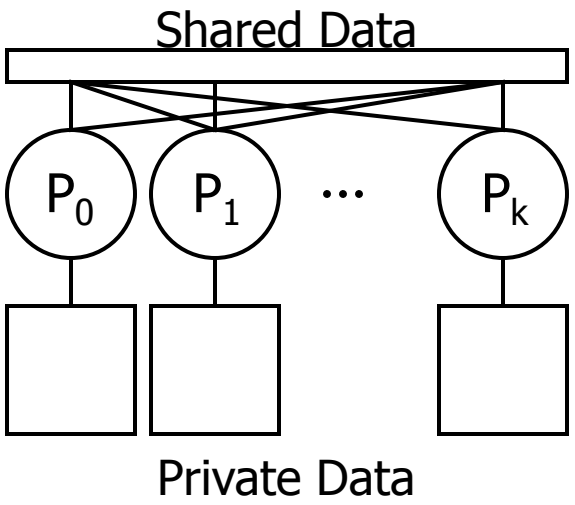
Message Passing Interface

- **Most widely used parallel programming model today**
- **Bindings for Fortran, C, C++, MATLAB**
- **P parallel processes, each with local data**
  - **MPI-1: Send/receive messages for inter-process communication**
  - **MPI-2: One-sided get/put data access from/to local data at remote process**
- **Explicit control of all inter-processor communication**
  - **Advantage: Programmer is conscious of communication overheads and attempts to minimize it**
  - **Drawback: Program development/debugging is tedious due to the partitioned-local view of the data**



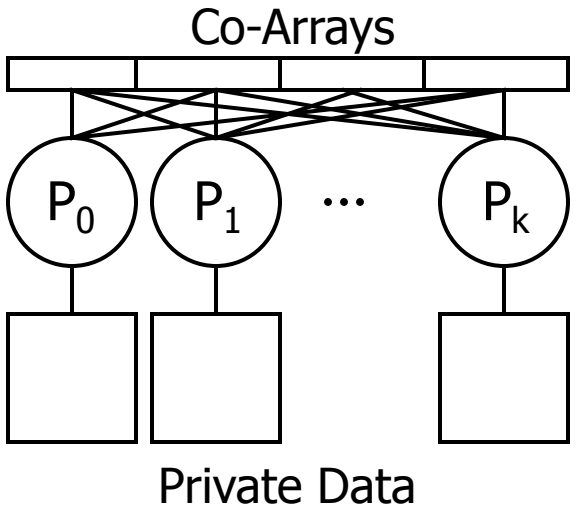
OpenMP

- **Uniform-Global view of shared data**
- **Available for Fortran, C, C++**
- **Work-sharing constructs (parallel loops and sections) and global-shared data view ease program development**
- **Disadvantage: Data locality issues obscured by programming model**



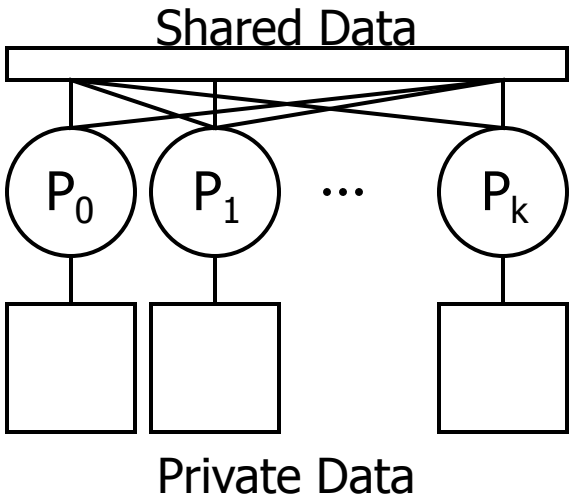
Co-Array Fortran

- **Partitioned, but global-shared data view**
- **SPMD programming model with local and shared variables**
- **Shared variables have additional co-array dimension(s), mapped to process space; each process can directly access array elements in the space of other processes**
  - **A(I,J) = A(I,J)[me-1] + A(I,J)[me+1]**
- **Compiler optimization of communication critical to performance, but all non-local access is explicit**



Unified Parallel C (UPC)

- **SPMD programming model with global shared view for arrays as well as pointer-based data structures**
- **Compiler optimizations critical for controlling inter-processor communication overhead**
  - **Very challenging problem since local vs. remote access is not explicit in syntax (unlike Co-Array Fortran)**
  - **Linearization of multidimensional arrays makes compiler optimization of communication very difficult**



Global Arrays vs. Other Models

Advantages:

- **Inter-operates with MPI**
  - **Use more convenient global-shared view for multi-dimensional arrays, but can use MPI model wherever needed**
- **Data-locality and granularity control is explicit with GA's get-compute-put model, unlike the non-transparent communication overheads with other models (except MPI)**
- **Library-based approach: does not rely upon smart compiler optimizations to achieve high performance**

Disadvantage:

- **Only useable for array data structures**

Overview of the Global Arrays Parallel Software Development Toolkit: Global Arrays Programming Model

Managed by UT-Battelle
for the Department of Energy

OAK RIDGE National Laboratory

Overview Of GA

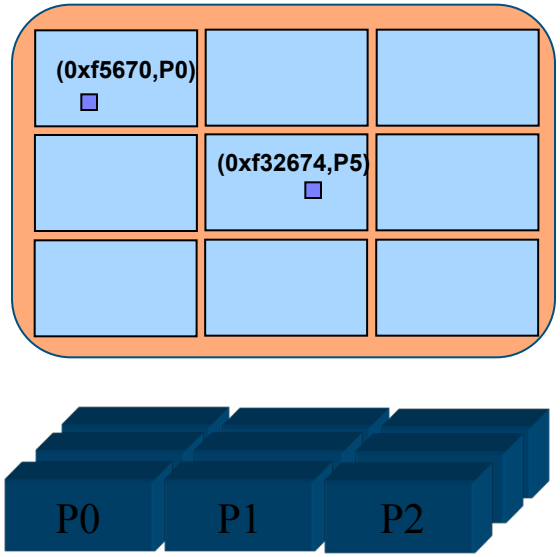
- **Programming model**
- **Structure of the GA toolkit**
- **Overview of interfaces**

Distributed vs Shared Data View

Distributed Data:

Data is explicitly associated with each processor, accessing data requires specifying the location of the data on the processor and the processor itself.

Data locality is explicit but data access is complicated. Distributed computing is typically implemented with message passing (e.g. MPI)

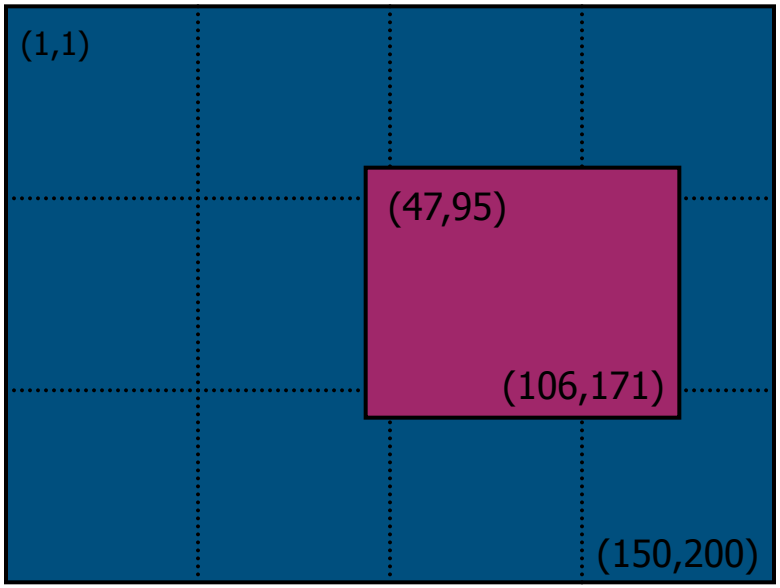


Distributed vs Shared Data (Cont).

Shared Data:

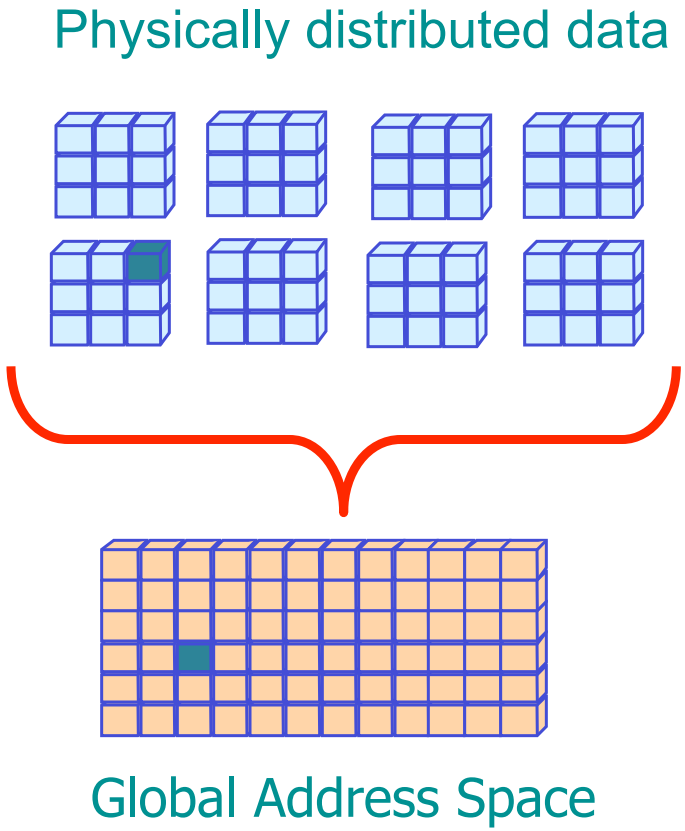
Data is in a globally accessible address space, any processor can access data by specifying its location using a global index

Data is mapped out in a natural manner (usually corresponding to the original problem) and access is easy. Information on data locality is obscured and leads to loss of performance.



Global Arrays

Distributed dense arrays that can be accessed through a shared data-like style



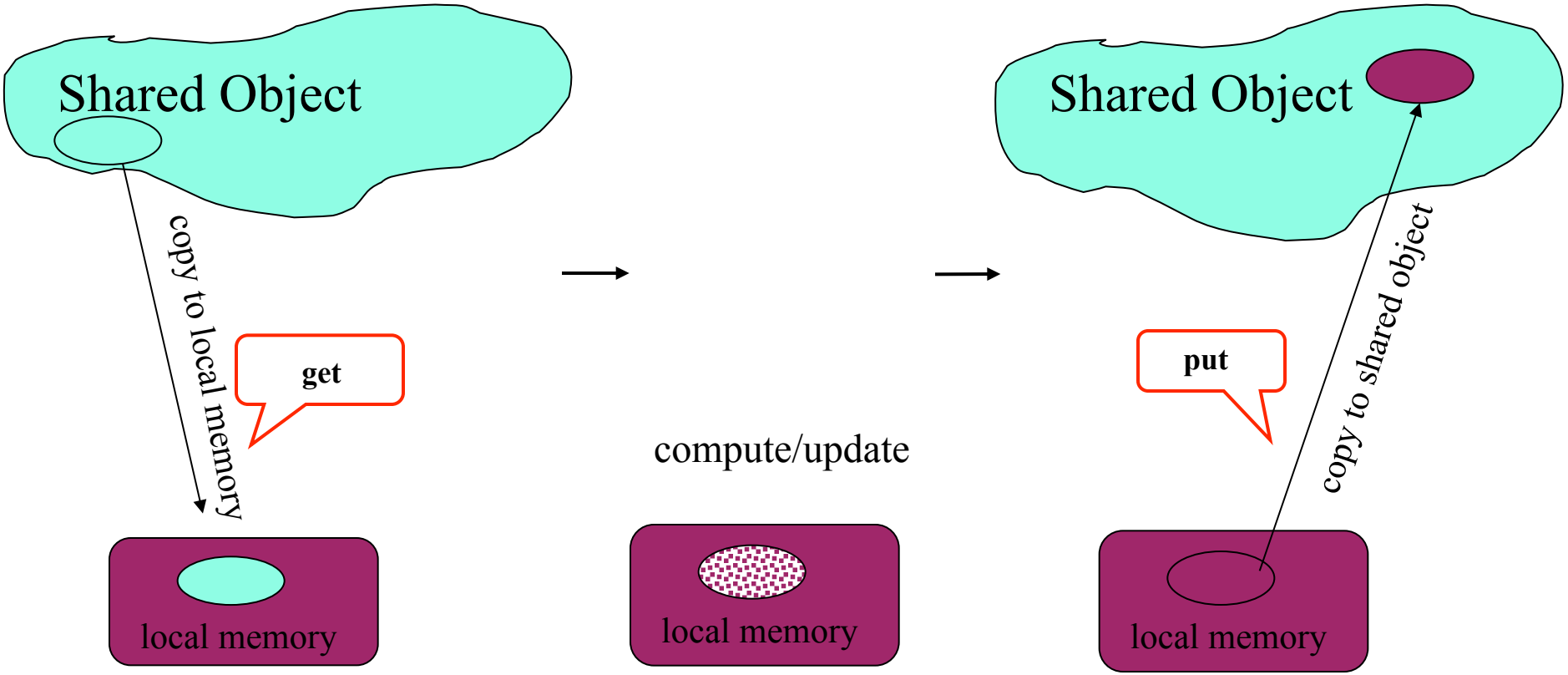
single, shared data structure/ global indexing

e.g., access A(4,3) rather than buf(7) on task 2

Global Arrays (cont.)

- **Shared data model in context of distributed dense arrays**
- **Much simpler than message-passing for many applications**
- **Complete environment for parallel code development**
- **Compatible with MPI**
- **Data locality control similar to distributed memory/message passing model**
- **Extensible**
- **Scalable**

Global Array Model of Computations



Creating Global Arrays

integer array handle

character string

minimum block size on each processor

```
g_a = NGA_Create(type, ndim, dims, name, chunk)
```

float, double, int, etc.

dimension

array of dimensions

Remote Data Access in GA vs MPI

Message Passing:

identify size and location of data blocks

loop over processors:
    if (me = P_N) then
        pack data in local message buffer
        send block of data to message buffer on P0
    else if (me = P0) then
        receive block of data from P_N in message buffer
        unpack data from message buffer to local buffer
    endif
end loop

copy local data on P0 to local buffer

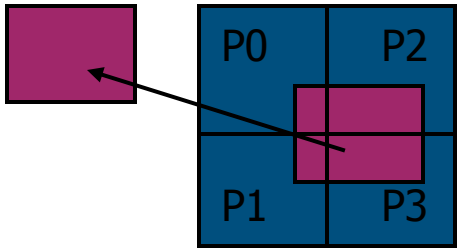
Global Arrays:

NGA_Get(g_a, lo, hi, buffer, ld);

Global Array handle

Global upper and lower indices of data patch

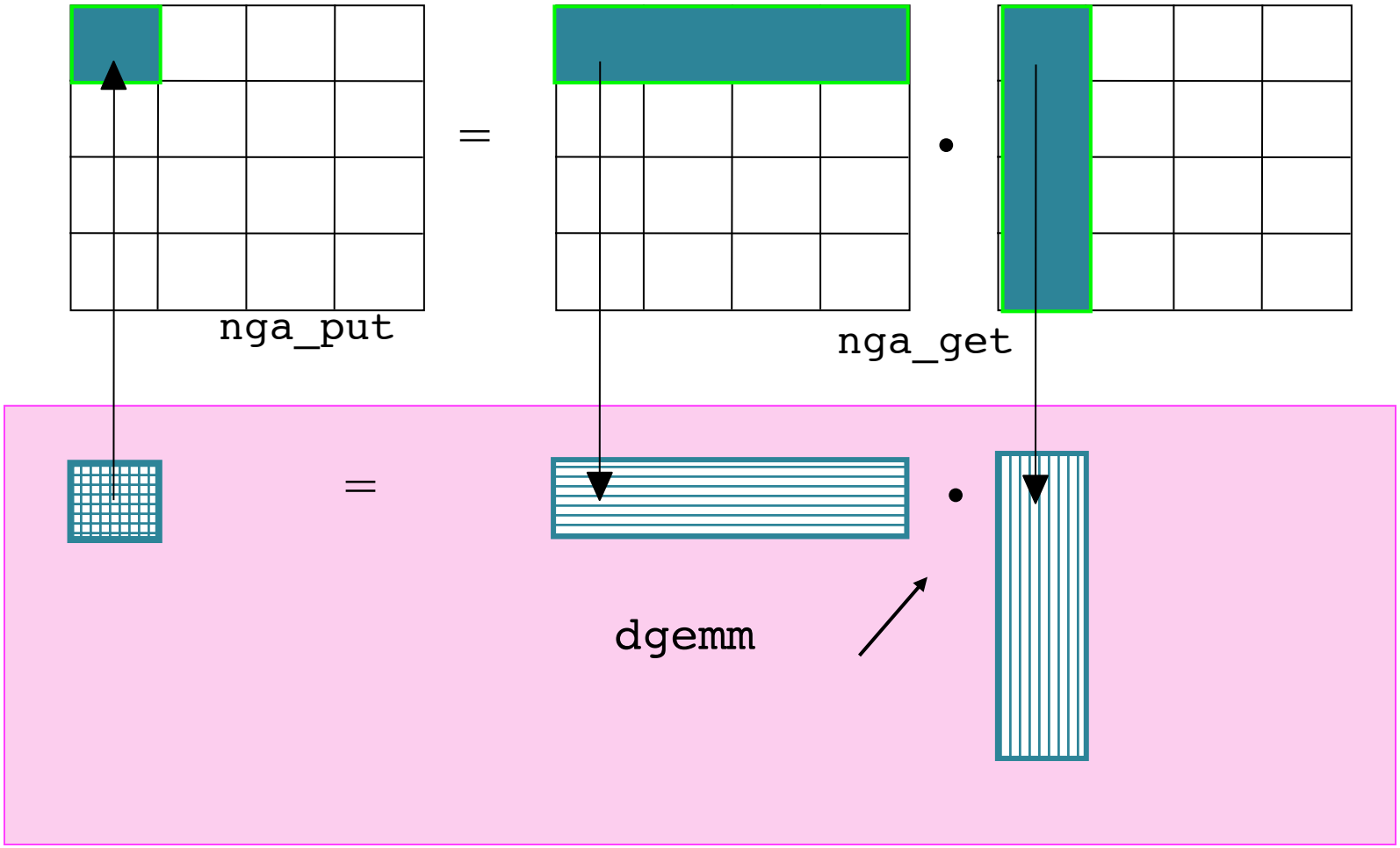
Local buffer and array of strides



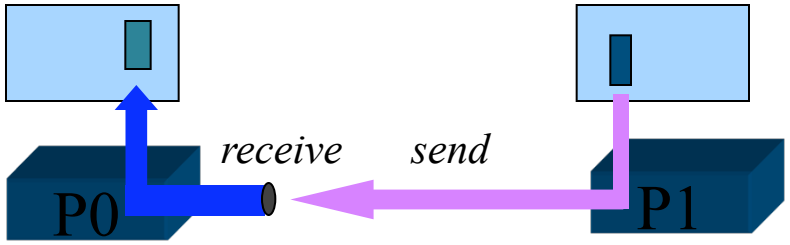
Example (invert an array)

```
/************ GA VERSION *************/
#define   NDIM       1
#define   TOTALELEMS   32768

int main(int argc, char **argv) {
   int dims,chunk,nprocs,me,i,lo,hi,lo2,hi2,ld;
   int g_a, g_b, a[TOTALELEMS],b[TOTALELEMS];

   GA_Initialize();
   me     = GA_Nodeid();
   nprocs = GA_Nnodes();
   dims   = nprocs*TOTALELEMS;
   chunk  = ld = TOTALELEMS;

   /* create a global array  */
   g_a = GA_Create(C_INT, NDIM, dims, "array A", chunk);
   g_b = GA_Duplicate(g_a, "array B");

   /* INITIALIZE DATA IN GA */

   GA_Distribution(g_a, me, lo, hi);
   GA_Get(g_a, lo, hi, a, ld);
   // INVERT DATA LOCALLY
   for (i=0; i<nelem; i++)    b[i] = a[nelem-1-i];
   // INVERT DATA GLOBALLY
   lo2 = dims - hi -1;
   hi2 = dims - lo -1;
   GA_Put(g_a,lo2,hi2,b,ld);

   GA_Terminate();
}
```

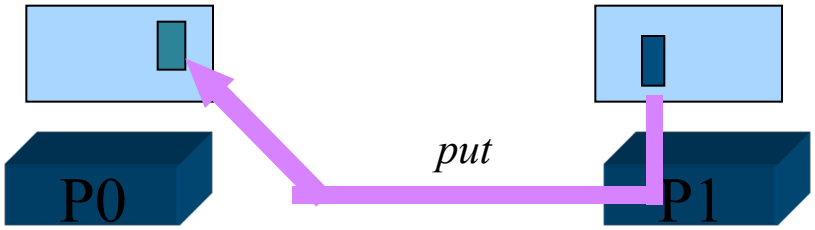
Matrix Multiplication



One-sided Communication



message passing
MPI



one-sided communication
SHMEM, ARMCI, MPI-2-1S

Message Passing:

Message requires cooperation on both sides. The processor sending the message (P1) and the processor receiving the message (P0) must both participate.

One-sided Communication:

Once message is initiated on sending processor (P1) the sending processor can continue computation. Receiving processor (P0) is not involved. Data is copied directly from switch into memory on P0.

Data Locality in GA

What data does a processor own?

```
NGA_Distribution(g_a, iproc, lo, hi);
```

Where is the data?

```
NGA_Access(g_a, lo, hi, ptr, ld)
```

Use this information to organize calculation so that maximum use is made of locally held data

Structure of GA

Application programming language interface

F90 Java

Fortran 77 C C++ Python Babel

distributed arrays layer
memory management, index translation

Global Arrays and MPI are completely interoperable. Code can contain calls to both libraries.

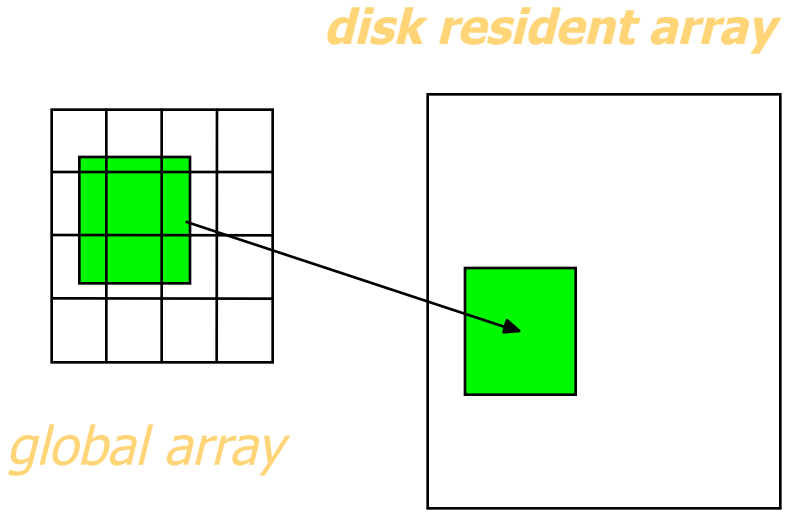
MPI
Global operations

ARMCI
portable 1-sided communication
put, get, locks, etc

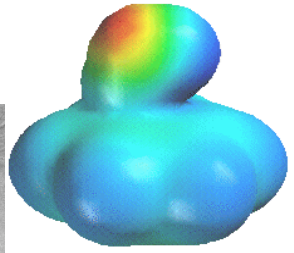
system specific interfaces
LAPI, GM/Myrinet, threads, VIA,..

Disk Resident Arrays

- **Extend GA model to disk**
  - **system similar to Panda (U. Illinois) but higher level APIs**
- **Provide easy transfer of data between N-dim arrays stored on disk and distributed arrays stored in memory**
- **Use when**
  - **Arrays too big to store in core**
  - **checkpoint/restart**
  - **out-of-core solvers**

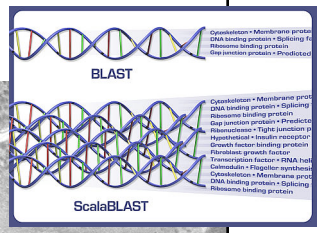


Application Areas

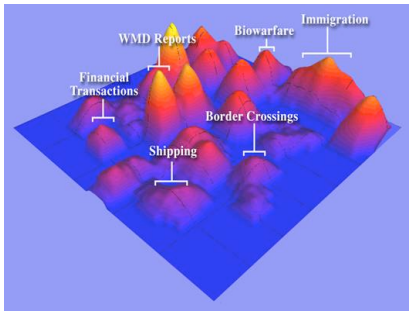


electronic structure chemistry

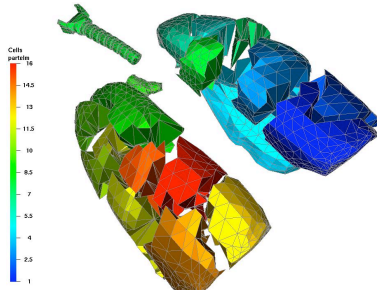
Major area



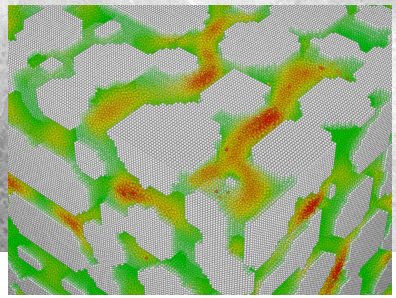
bioinformatics



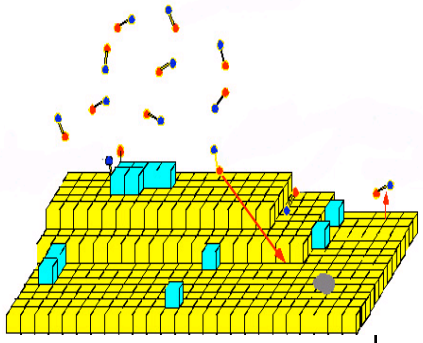
visual analytics



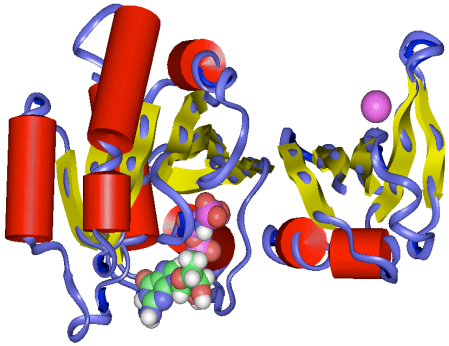
Biology: organ simulation



smooth particle hydrodynamics



material sciences

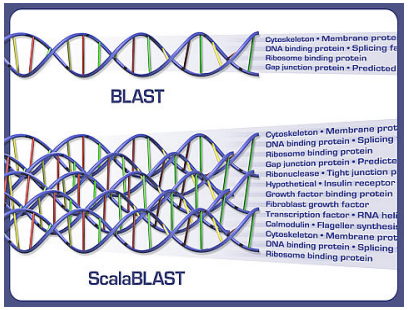


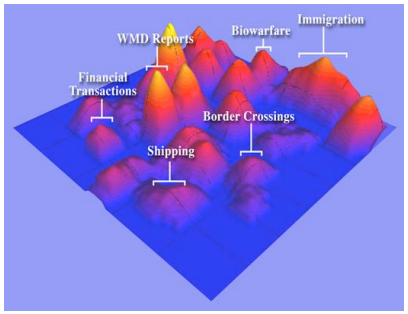
molecular dynamics

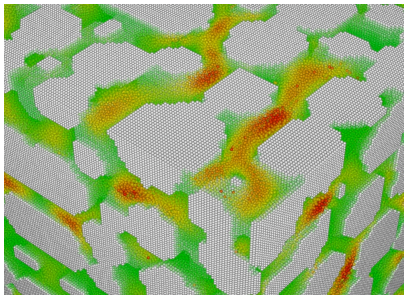
Visualization and image analysis

Others: financial security forecasting, astrophysics, geosciences, atmospheric chemistry

New Applications







ScalaBLAST

C. Oehmen and J. Nieplocha. ScalaBLAST: "A scalable implementation of BLAST for high performance data-intensive bioinformatics analysis." IEEE Trans. Parallel Distributed Systems, Vol. 17, No. 8, 2006

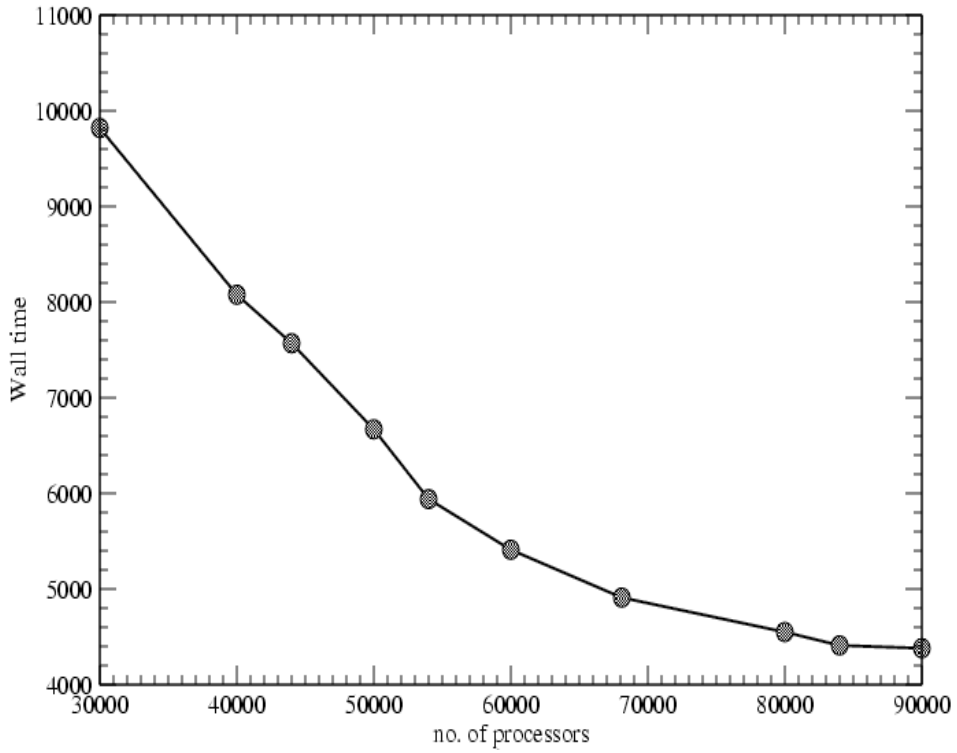
Parallel Inspire

Krishnan M, SJ Bohn, WE Cowley, VL Crow, and J Nieplocha, "Scalable Visual Analytics of Massive Textual Datasets", Proc. IEEE International Parallel and Distributed Processing Symposium, 2007.

Smooth Particle Hydrodynamics

Productivity and Scalability

- **Liquid Water: Obtaining the Right Answer for the Right Reasons**
  - **Edoardo Apra, Robert J. Harrison, Wibe A. de Jong, Alistair Rendell, Vinod Tipparaju, Sotiris Xantheas**
- **SC2009 Gordon Bell Finalist**



$(H_2O)_{18}$ CCSD(T)benchmarkonXT5

Source Code and More Information

- Version 4.1 available
- Homepage at http://www.emsl.pnl.gov/docs/global/
- Platforms (32 and 64 bit)
  - IBM SP, BlueGene
  - Cray X1, XD1, XT3, XT4, XT5
  - Linux Cluster with Ethernet, Myrinet, Infiniband, or Quadrics
  - Solaris
  - Fujitsu
  - Hitachi
  - NEC
  - HP
  - Windows

Overview of the Global Arrays Parallel Software Development Toolkit: Getting Started, Basic Calls

Managed by UT-Battelle
for the Department of Energy

OAK RIDGE National Laboratory

Outline

- **Writing, Building, and Running GA Programs**
- **Basic Calls**
- **Intermediate Calls**
- **Advanced Calls**

Writing, Building and Running GA programs

- Installing GA
- Writing GA programs
- Compiling and linking
- Running GA programs
- For detailed information
  - GA Webpage
    - GA papers, APIs, user manual, etc.
    - (Google: Global Arrays)
    - http://www.emsl.pnl.gov/docs/global/
  - GA User Manual
    - http://www.emsl.pnl.gov/docs/global/user.html
  - GA API Documentation
    - GA Webpage => User Interface
    - http://www.emsl.pnl.gov/docs/global/userinterface.html
  - GA Support/Help
    - hpctools@pnl.gov or hpctools@emsl.pnl.gov
  - 2 mailing lists: GA User Forum, and GA Announce

Installing GA

- Required environment settings
  - TARGET: Used to set the platform
    - E.g. setenv TARGET LINUX (32 bit Linux platform)
    - See chapter 2 of GA user manual for the complete list
  - ARMCI_NETWORK: Specify the underlying network communication protocol
    - This setting is required only on clusters with a high performance network
    - E.g. If the underlying network is Infiniband using OpenIB protocol
      - setenv ARMCI_NETWORK OPENIB
  - GA requires MPI for basic start-up and process management
    - You can either use MPI or TCGMSG wrapper to MPI.
      - To use MPI, setenv MSG_COMMS MPI
      - To use TCGMSG-MPI wrapper, setenv USE_MPI y
    - Also set MPI_LIB and MPI_INCLUDE, which contain the path to MPI include and libraries
    - Set LIBMPI which point to the actual MPI libs e.g. setenv LIBMPI -lmpich
- Please refer to chapter 2 of user manual for other optional arguments
- "make" or "gmake" to build GA libraries
- To override the default compiler and optimization flags when building GA:
  - gmake FC=f90 CC=cc FOPT=-O4 COPT=-g

Writing GA Programs

- **GA Definitions and Data types**
  - **C programs include files: ga.h, macdecls.h**
  - **Fortran programs should include the files: mafdecls.fh, global.fh.**
- **GA Initialize, GA_Terminate --> initializes and terminates GA library**

```
#include <stdio.h>
#include "mpi.h"
#include "ga.h"
#include "macdecls.h"

int main( int argc, char **argv ) {
        MPI_Init( &argc, &argv );
        GA_Initialize();

        printf( "Hello world\n" );

        GA_Terminate();
        MPI_Finalize();
        return 0;
}
```

Writing GA Programs

- GA requires the following functionalities from a message passing library (MPI/TCGMSG)
  - initialization and termination of processes
  - Broadcast, Barrier
  - a function to abort the running parallel job in case of an error
- The message-passing library has to be
  - initialized before the GA library
  - terminated after the GA library is terminated
- GA is compatible with MPI

```
#include <stdio.h>
#include "mpi.h"
#include "ga.h"
#include "macdecls.h"

int main( int argc, char **argv ) {
          MPI_Init( &argc, &argv );
          GA_Initialize();

          printf( "Hello world\n" );

          GA_Terminate();
          MPI_Finalize();
          return 0;
}
```

Compiling and Linking GA Programs

- **2 ways**
  - **Use the GA Makefile in global/testing**
  - **Your Makefile**

- **GA Makefile in global/testing**
  - **To compile and link your GA based program, for example "app.c" (or "app.f", ..)**
    - **Copy to $GA_DIR/global/testing, and type**
    - **make app.x or gmake app.x**
  - **Compile any test program in GA testing directory, and use the appropriate compile/link flags in your program**

Compiling and Linking GA Programs (cont.)

- **Your Makefile**
  - **Please refer to the INCLUDES, FLAGS and LIBS variables, which will be printed at the end of a successful GA installation on your platform**

```
INCLUDES = -I./include -I/msrc/apps/mpich-1.2.6/gcc/ch_shmem/include
LIBS     = -L/msrc/home/manoj/GA/cvs/lib/LINUX -lglobal -lma  -llinalg
           -larmci -L/msrc/apps/mpich-1.2.6/gcc/ch_shmem/lib -lmpich -lm

For Fortran Programs:
FLAGS    = -g  -Wall -funroll-loops -fomit-frame-pointer -malign-double
           -fno-second-underscore -Wno-globals

For C Programs:
FLAGS    = -g  -Wall -funroll-loops -fomit-frame-pointer -malign-double
           -fno-second-underscore -Wno-globals
```

  - **You can use these variables in your Makefile**
    - **For example: gcc $(INCLUDES) $(FLAGS) –o ga_test ga_test.c $(LIBS)**

- NOTE: Please refer to GA user manual chapter 2 for more information

Running GA Programs

- **Example: Running a test program "ga_test" on 2 processes**
- **mpirun -np 2 ga_test**
- **Running a GA program is same as running an MPI program**

Outline

- **Writing, Building, and Running GA Programs**
- **Basic Calls**
- **Intermediate Calls**
- **Advanced Calls**

GA Basic Operations

- GA programming model is very simple.
- Most of a parallel program can be written with these basic calls
  - GA_Initialize, GA_Terminate
  - GA_Nnodes, GA_Nodeid
  - GA_Create, GA_Destroy
  - GA_Put, GA_Get
  - GA_Sync

GA Initialization/Termination

- There are two functions to *initialize* GA:
  - Fortran
    - subroutine ga_initialize()
    - subroutine ga_initialize_ltd(limit)
  - C
    - void GA_Initialize()
    - void GA_Initialize_ltd(size_t limit)

- To *terminate* a GA program:
  - Fortran subroutine ga_terminate()
  - Cvoid GA_Terminate()

```
      program main
      #include "mafdecls.h"
      #include "global.fh"
      integer ierr
c
      call mpi_init(ierr)
      call ga_initialize()
c
      write(6,*) 'Hello world'
c
      call ga_terminate()
      call mpi_finilize()
      end
```

integer limit - amount of memory in bytes per process [input]

Parallel Environment - Process Information

- **Parallel Environment:**
  - **how many processes are working together (*size*)**
  - **what their IDs are (ranges from 0 to *size*-1)**
- **To return the process ID of the current process:**
  - **Fortran integer function ga_nodeid()**
  - **C int GA_Nodeid()**
- **To determine the number of computing processes:**
  - **Fortran integer function ga_nnodes()**
  - **C int GA_Nnodes()**

Parallel Environment – Process Information (EXAMPLE)

```
program main
#include "mafdecls.h"
#include "global.fh"
integer ierr,me,nproc

call mpi_init(ierr)
call ga_initialize()

me = ga_nodeid()
size = ga_nnodes()
write(6,*) 'Hello world: My rank is ' +  me + ' out of ' +
!                size + 'processes/nodes'

call ga_terminate()
call mpi_finilize()
end
```

$ mpirun –np 4 helloworld

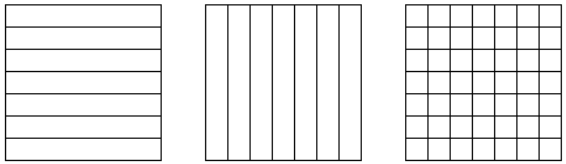
Hello world: My rank is 0 out of 4 processes/nodes
Hello world: My rank is 2 out of 4 processes/nodes
Hello world: My rank is 3 out of 4 processes/nodes
Hello world: My rank is 1 out of 4 processes/nodes

GA Data Types

- C Data types
  - C_INT - int
  - C_LONG - long
  - C_FLOAT - float
  - C_DBL - double
  - C_SCPL - single complex
  - C_DCPL - double complex
- Fortran Data types
  - MT_F_INT - integer (4/8 bytes)
  - MT_F_REAL - real
  - MT_F_DBL - double precision
  - MT_F_SCPL - single complex
  - MT_F_DCPL - double complex

Creating/Destroying Arrays

- To *create* an array with a regular distribution:
  - Fortran logical function nga_create(type, ndim, dims, name, chunk, g_a)
  - C int NGA_Create(int type, int ndim, int dims[], char *name, int chunk[])

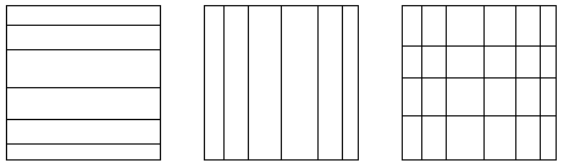


character*(*)	name	- a unique character string	[input]
integer	type	- GA data type	[input]
integer	dims()	- array dimensions	[input]
integer	chunk()	- minimum size that dimensions should be chunked into	[input]
integer	g_a	- array handle for future references	[output]

```
      dims(1) = 5000
      dims(2) = 5000
      chunk(1) = -1      !Use defaults
      chunk(2) = -1
      if (.not.nga_create(MT_F_DBL,2,dims,'Array_A',chunk,g_a))
     +    call ga_error("Could not create global array A",g_a)
```

Creating/Destroying Arrays (cont.)

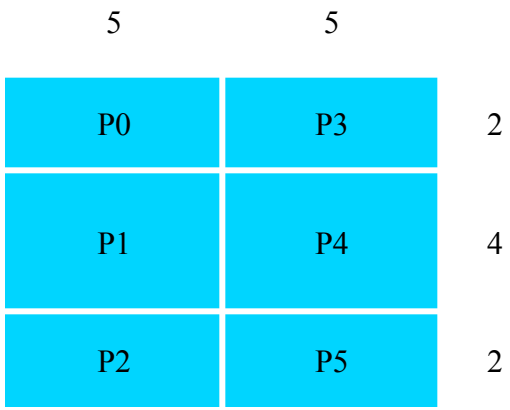
- **To *create* an array with an irregular distribution:**
  - **Fortran logical function nga_create_irreg (type, ndim, dims, array_name, map, nblock, g_a)**
  - **C int NGA_Create_irreg(int type, int ndim, int dims[], char* array_name, nblock[], map[])**



character*(*)	name	- a unique character string	[input]
integer	type	- GA datatype	[input]
integer	dims	- array dimensions	[input]
integer	nblock(*)	- no. of blocks each dimension is divided into	[input]
integer	map(*)	- starting index for each block	[input]
integer	g_a	- integer handle for future references	[output]

Creating/Destroying Arrays (cont.)

- **Example of irregular distribution:**
  - **The distribution is specified as a Cartesian product of distributions for each dimension. The array indices start at 1.**
    - **The figure demonstrates distribution of a 2-dimensional array 8x10 on 6 (or more) processors. *block[2]*={3,2}, the size of *map* array is *s*=5 and array *map* contains the following elements *map*={1,3,7, 1, 6}.**
    - **The distribution is nonuniform because, P1 and P4 get 20 elements each and processors P0,P2,P3, and P5 only 10 elements each.**




```
block(1) = 3
block(2) = 2
map(1) = 1
map(2) = 3
map(3) = 7
map(4) = 1
map(5) = 6
if (.not.nga_create_irreg(MT_F_DBL,2,dims,'Array_A',map,block,g_a))
+    call ga_error("Could not create global array A",g_a)
```

Creating/Destroying Arrays (cont.)

- To *duplicate* an array:
  - Fortran logical function ga_duplicate(g_a, g_b, name)
  - C int GA_Duplicate(int g_a, char *name)

- Global arrays can be *destroyed* by calling the function:
  - Fortran subroutine ga_destroy(g_a)
  - C void GA_Destroy(int g_a)

```
 call nga_create(MT_F_INT,dim,dims,
+     'array_a',chunk,g_a)
 call ga_duplicate(g_a,g_b,'array_b')
 call ga_destroy(g_a)
```

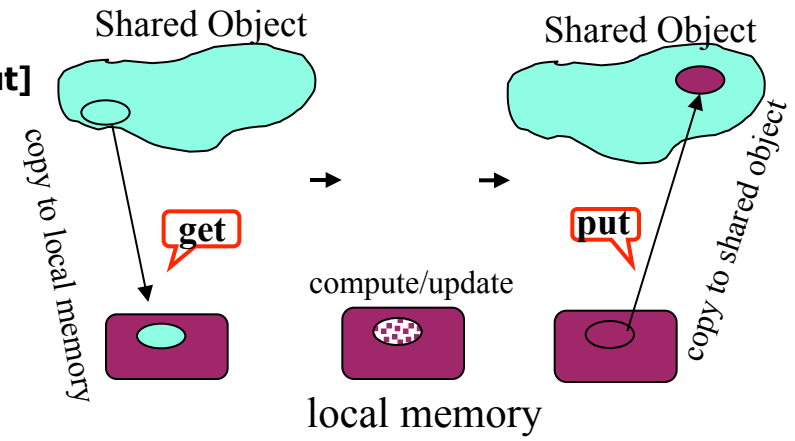
integer g_a, g_b
character*(*) name

name	- a character string	[input]
g_a	- Integer handle for reference array	[input]
g_b	- Integer handle for new array	[output]

Put/Get

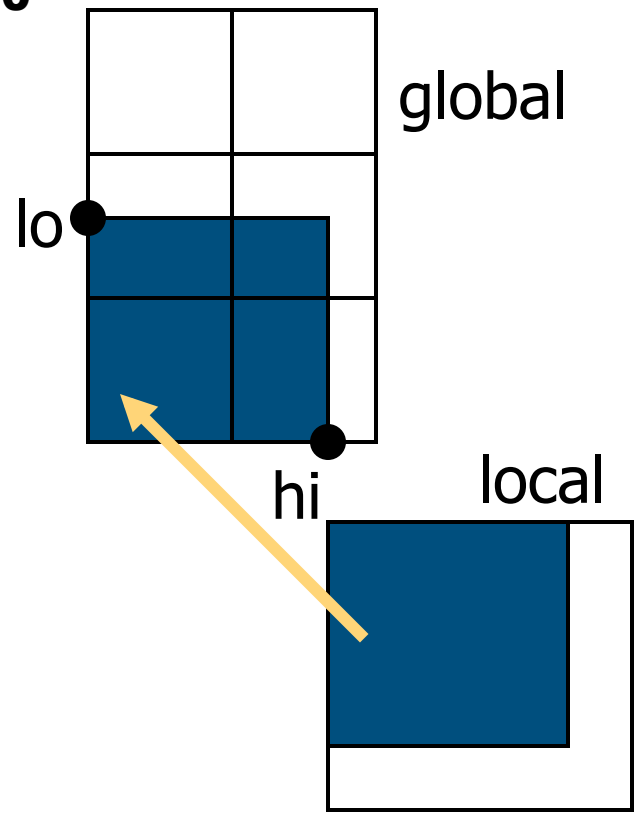
- ***Put*** **copies data from a local array to a global array section:**
  - **Fortran subroutine nga_put(g_a, lo, hi, buf, ld)**
  - **C void NGA_Put(int g_a, int lo[], int hi[], void *buf, int ld[])**
- ***Get*** **copies data from a global array section to a local array:**
  - **Fortran subroutine nga_get(g_a, lo, hi, buf, ld)**
  - **C void NGA_Get(int g_a, int lo[], int hi[], void *buf, int ld[])**

integer	**g_a global array handle [input]**
integer	**lo(),hi() limits on data block to be moved [input]**
Double	**precision/complex/integer buf local buffer [output]**
integer	**ld() array of strides for local buffer [input]**



Put/Get (cont.)

- **Example of *put* operation:**
  - **transfer data from a local buffer (10 x10 array) to (7:15,1:8) section of a 2-dimensional 15 x10 global array into *lo*={7,1}, *hi*={15,8}, ld={10}**

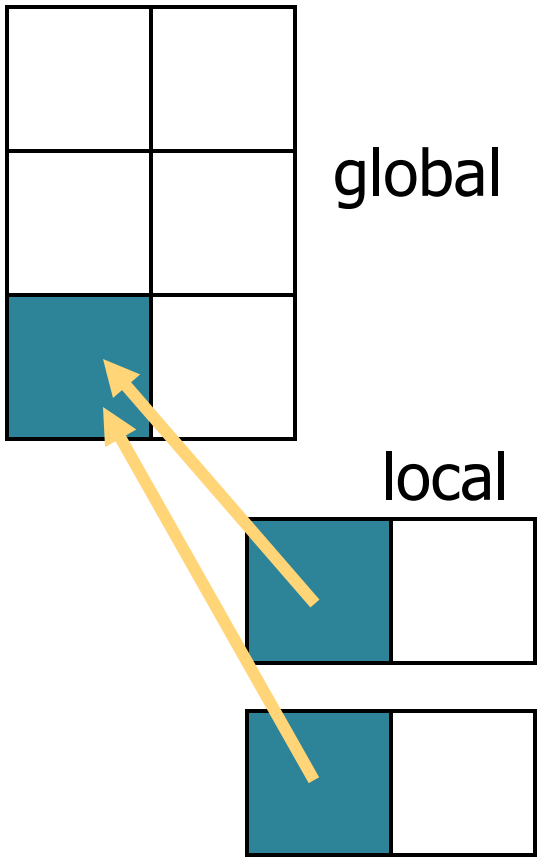



```
double precision buf(10,10)
  :
  :
call nga_put(g_a,lo,hi,buf,ld)
```

Atomic Accumulate

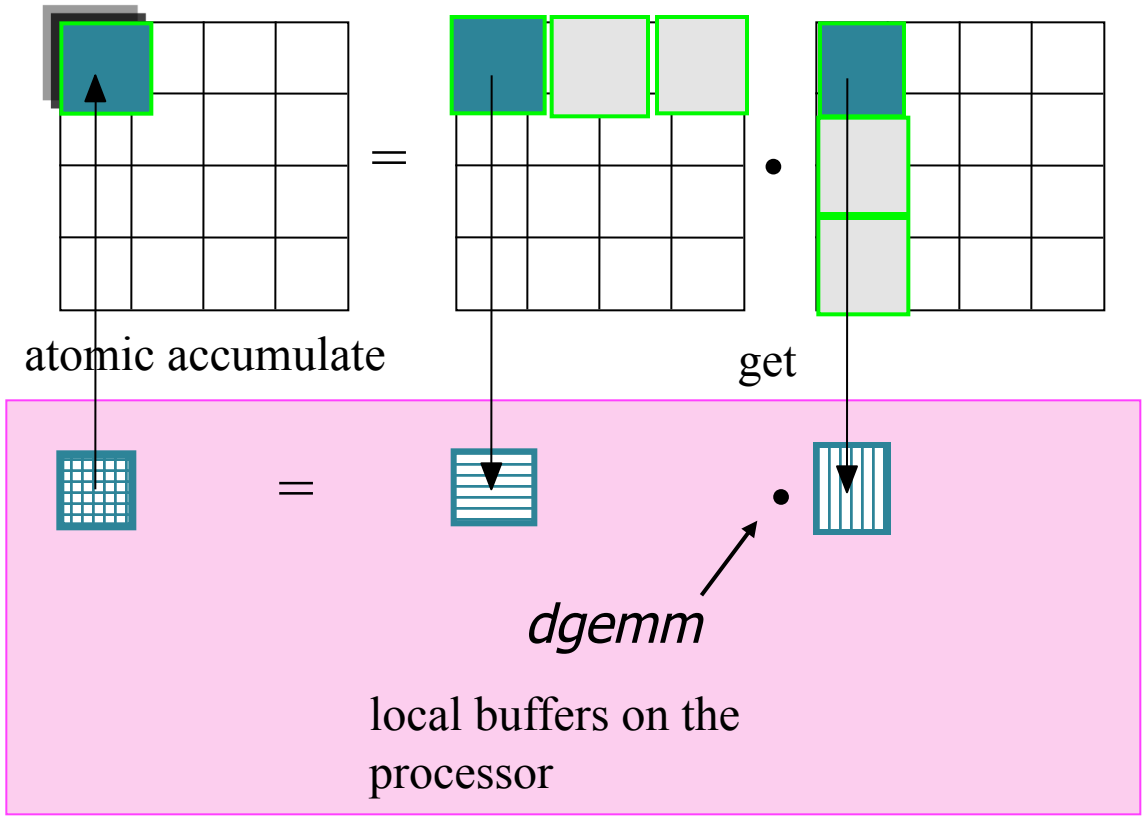
- ***Accumulate*** **combines the data from the local array with data in the global array section:**
  - **Fortran -- subroutine nga_acc(g_a, lo, hi,buf, ld, alpha)**
  - **C -- void NGA_Acc(int g_a, int lo[], int hi[], void *buf, int ld[], void *alpha)**

integer	**g_a array handle**	**[input]**
integer	**lo(), hi() limits on data block to be moved**	**[input]**
double	**precision/complex buf local buffer**	**[input]**
integer	**ld() array of strides for local buffer**	**[input]**
double	**precision/complex alpha arbitrary scale factor**	**[input]**



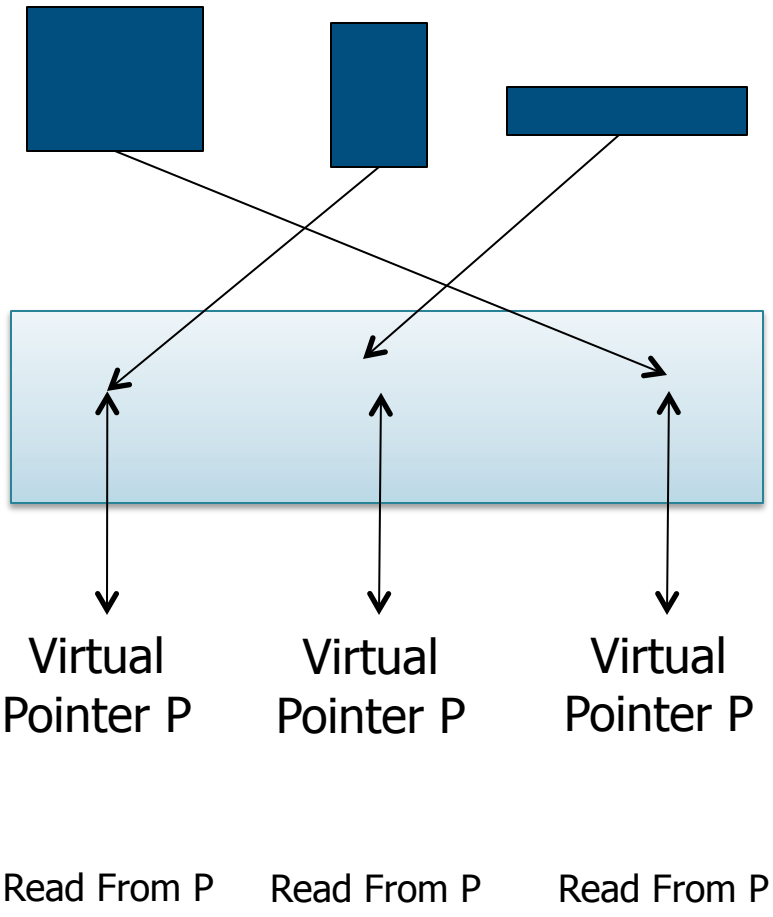
ga(i,j) = ga(i,j)+alpha*buf(k,l)

Matrix Multiply



more scalable!
(less memory,
higher parallelism)

What's wrong with this picture?



GA_Put(Mysection, GA)

MPI_Barrier

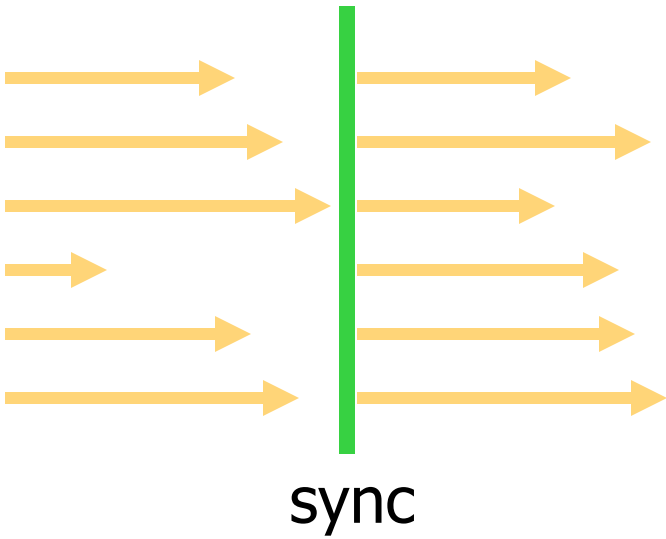
GA_Access(GA,P)

Read P(i)

Sync

- ***Sync*** **is a collective operation**
- **It acts as a barrier, which synchronizes all the processes and ensures that all the Global Array operations are complete at the call**

- **The functions are:**
  - **Fortran subroutine ga_sync()**
  - **C void GA_Sync()**



Global Operations

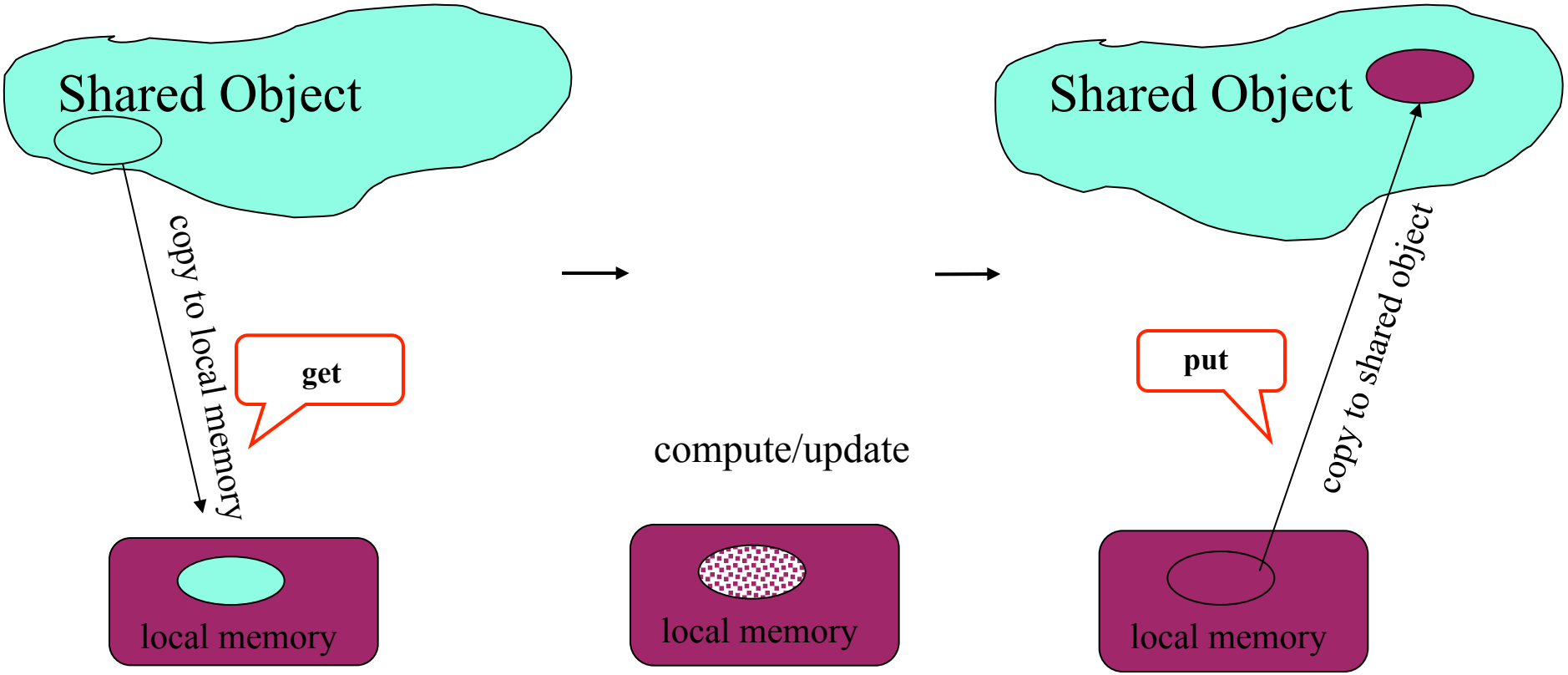
- **Fortran**

```
subroutine ga_brdcst(type, buf, lenbuf, root)

subroutine ga_igop(type, x, n, op)
subroutine ga_dgop(type, x, n, op)
```

- **C**

```
void GA_Brdcst(void *buf, int lenbuf, int root)

void GA_Igop(long x[], int n, char *op)

void GA_Dgop(double x[], int n, char *op)
```

Global Array Model of Computations



Locality Information

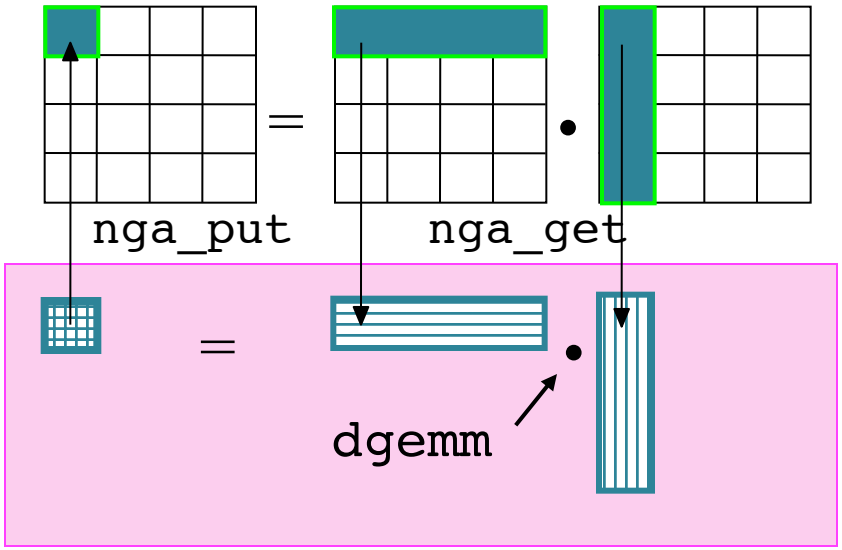
- **Discover array elements held by each processor**
  - **Fortran** nga_distribution(g_a,proc,lo,hi)
  - **C** void NGA_Distribution(int g_a, int proc, int *lo, int*hi)

integer	g_a	array handle	[input]
integer	proc	processor ID	[input]
integer	lo(ndim)	lower index	[output]
integer	hi(ndim)	upper index	[output]

```
do iproc = 1, nproc
  write(6,*) 'Printing g_a info for processor',iproc
  call nga_distribution(g_a,iproc,lo,hi)
  do j = 1, ndim
    write(6,*) j,lo(j),hi(j)
  end do
dnd do
```

Example: Matrix Multiply

```
/* Determine which block of data is locally owned. Note that
   the same block is locally owned for all GAs. */
NGA_Distribution(g_c, me, lo, hi);
/* Get the blocks from g_a and g_b needed to compute this block in
   g_c and copy them into the local buffers a and b. */
lo2[0] = lo[0]; lo2[1] = 0; hi2[0] = hi[0]; hi2[1] = dims[0]-1;
NGA_Get(g_a, lo2, hi2, a, ld);
lo3[0] = 0; lo3[1] = lo[1]; hi3[0] = dims[1]-1; hi3[1] = hi[1];
NGA_Get(g_b, lo3, hi3, b, ld);
/* Do local matrix multiplication and store the result in local
   buffer c. Start by evaluating the transpose of b. */
for(i=0; i < hi3[0]-lo3[0]+1; i++)
  for(j=0; j < hi3[1]-lo3[1]+1; j++)
     btrns[j][i] = b[i][j];
/* Multiply a and b to get c */
for(i=0; i < hi[0] - lo[0] + 1; i++) {
  for(j=0; j < hi[1] - lo[1] + 1; j++) {
    c[i][j] = 0.0;
    for(k=0; k<dims[0]; k++)
      c[i][j] = c[i][j] + a[i][k]*btrns[j][k];
  }
}
/* Copy c back to g_c */
NGA_Put(g_c, lo, hi, c, ld);
```



Overview of the Global Arrays Parallel Software Development Toolkit: Intermediate and Advanced APIs

Managed by UT-Battelle
for the Department of Energy

OAK RIDGE National Laboratory

Outline

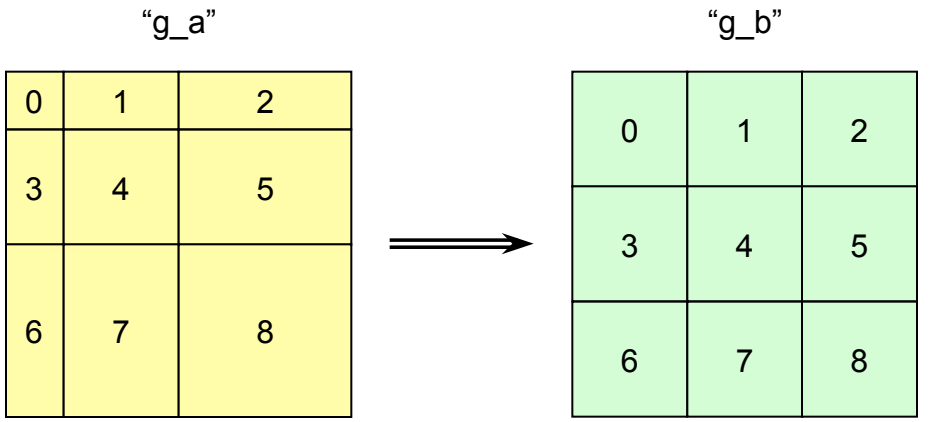
- Writing, Building, and Running GA Programs
- Basic Calls
- Intermediate Calls
- Writing Scalable GA code with Advanced Calls

Basic Array Operations

- **Whole Arrays:**
  - **To set all the elements in the array to zero:**
    - **Fortran** **subroutine ga_zero(g_a)**
    - **C** **void GA_Zero(int g_a)**
  - **To assign a single value to all the elements in array:**
    - **Fortran** **subroutine ga_fill(g_a, val)**
    - **C** **void GA_Fill(int g_a, void *val)**
  - **To scale all the elements in the array by factor *val*:**
    - **Fortran** **subroutine ga_scale(g_a, val)**
    - **C** **void GA_Scale(int g_a, void *val)**

Basic Array Operations (cont.)

- **Whole Arrays:**
  - **To copy data between two arrays:**
    - **Fortran** **subroutine ga_copy(g_a, g_b)**
    - **C** **void GA_Copy(int g_a, int g_b)**
    - **Arrays must be same size and dimension**
      - **Distribution may be different**

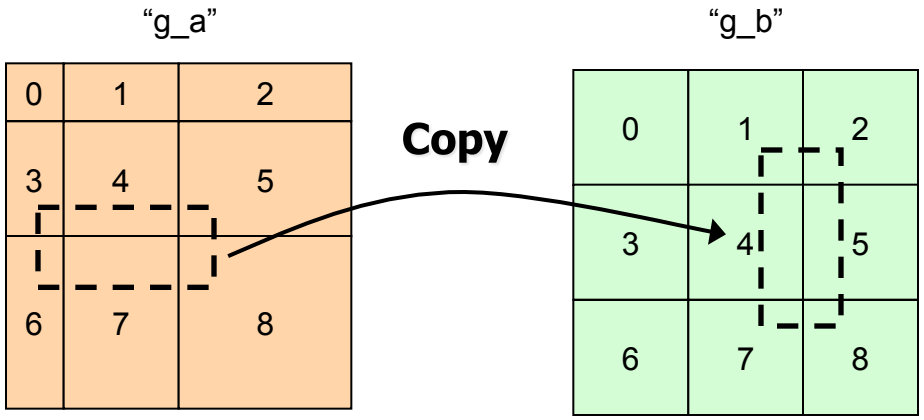



```
call ga_create(MT_F_INT,ndim,dims,
     'array_A',chunk_a,g_a)
call nga_create(MT_F_INT,ndim,dims,
     'array_B',chunk_b,g_b)

    ... Initialize g_a ....

call ga_copy(g_a, g_b)
```

Global Arrays g_a and g_b distributed on a 3x3 process grid

Basic Array Operations (cont.)

- **Patch Operations:**
  - **The copy patch operation:**
    - **Fortran - subroutine nga_copy_patch(trans, g_a, alo, ahi, g_b, blo, bhi)**
    - **C - void NGA_Copy_patch(char trans, int g_a, int alo[], int ahi[], int g_b, int blo[], int bhi[])**
    - **Number of elements must match**



Basic Array Operations (cont.)

- **Patches (Cont):**
  - **To set only the region defined by *lo* and *hi* to zero:**
    - **Fortran** **subroutine nga_zero_patch(g_a, lo, hi)**
    - **C** **void NGA_Zero_patch(int g_a, int lo[] int hi[])**
  - **To assign a single value to all the elements in a patch:**
    - **Fortran** **subroutine nga_fill_patch(g_a, lo, hi, val)**
    - **C** **voidNGA_Fill_patch(int g_a, int lo[] int hi[], void *val)**
  - **To scale the patch defined by lo and hi by the factor val:**
    - **Fortran** **subroutine nga_scale_patch(g_a, lo, hi, val)**
    - **C** **voidNGA_Scale_patch(int g_a, int lo[] int hi[], void *val)**
  - **The copy patch operation:**
    - **Fortran** **subroutine nga_copy_patch(trans, g_a, alo, ahi, g_b, blo, bhi)**
    - **C** **void NGA_Copy_patch(char trans, int g_a, int alo[], int ahi[], int g_b, int blo[], int bhi[])**

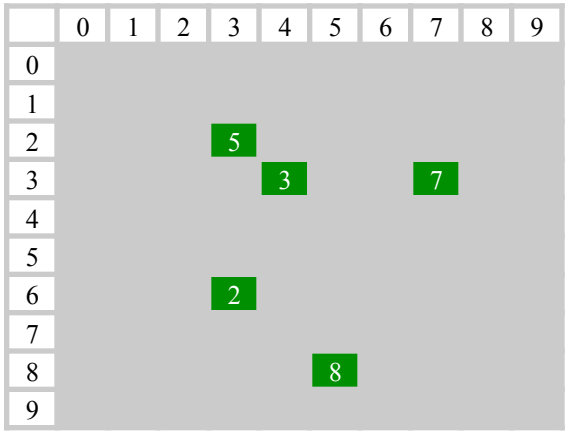
Scatter/Gather

- ***Scatter*** **puts array elements into a global array:**
  - **Fortran - subroutine nga_scatter(g_a, v, subscrpt_array, n)**
  - **C - void NGA_Scatter(int g_a, void *v, int *subscrpt_array[], int n)**
- ***Gather*** **gets the array elements from a global array into a local array:**
  - **Fortran - subroutine nga_gather(g_a, v, subscrpt_array, n)**
  - **C - void NGA_Gather(int g_a, void *v, int *subscrpt_array[], int n)**

integer	g_a array handle	[input]
double precision	v(n) array of values	[input/output]
integer	n number of value	[input]
integer	subscrpt_array location of values in global array	[input]

Scatter/Gather (cont.)

- **Example of *scatter* operation:**
  - **Scatter the 5 elements into a 10x10 global array**
    - **Element 1** **v[0] = 5**
      subsArray[0][0] = 2
      subsArray[0][1] = 3
    - **Element 2** **v[1] = 3**
      subsArray[1][0] = 3
      subsArray[1][1] = 4
    - **Element 3** **v[2] = 8**
      subsArray[2][0] = 8
      subsArray[2][1] = 5
    - **Element 4** **v[3] = 7**
      subsArray[3][0] = 3
      subsArray[3][1] = 7
    - **Element 5** **v[4] = 2**
      subsArray[4][0] = 6
      subsArray[4][1] = 3
  - **After the *scatter* operation, the five elements would be scattered into the global array as shown in the figure.**

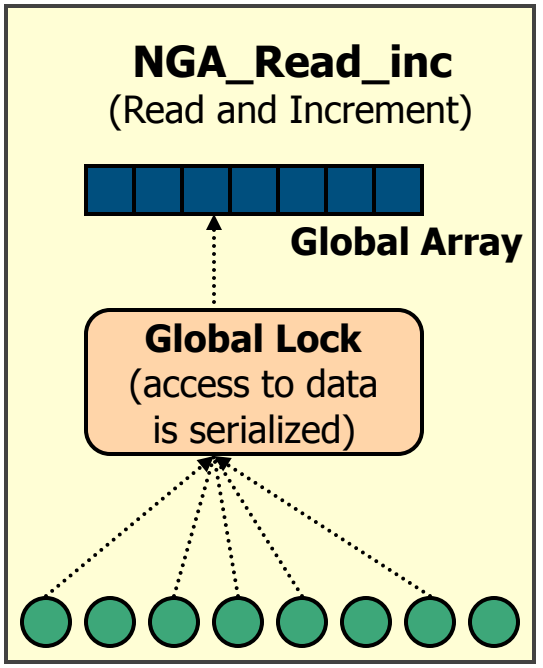



```
integer subscript(ndim,nlen)
   :
call nga_scatter(g_a,v,subscript,nlen)
```

Read and Increment

- ***Read_inc*** **remotely updates a particular element in an integer global array and returns the original value:**
  - **Fortran integer function nga_read_inc(g_a, subscript, inc)**
  - **C long NGA_Read_inc(int g_a, int subscript[], long inc)**
  - **Applies to integer arrays only**
  - **Can be used as a global counter for dynamic load balancing**

integer	g_a	[input]
integer	subscript(ndim), inc	[input]

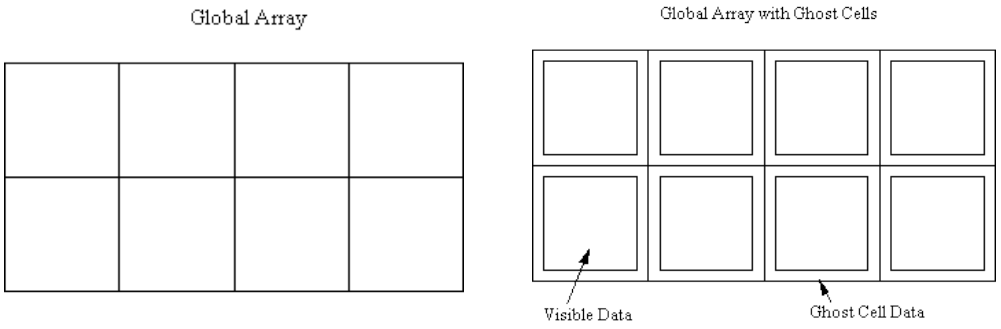



```
c  Create task counter
   call nga_create(MT_F_INT,one,one,chunk,g_counter)
   call ga_zero(g_counter)
         :
   itask = nga_read_inc(g_counter,one,one)

   ... Translate itask into task ...
```

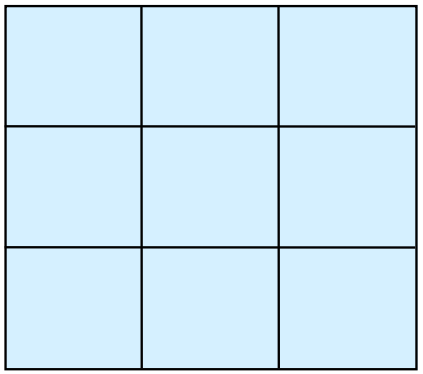
Creating Arrays with Ghost Cells

- **To create arrays with ghost cells:**
  - **For arrays with regular distribution:**
    - **Fortran** — **logical function nga_create_ghosts(type, dims, width, array_name, chunk, g_a)**
    - **C** — **int NGA_Create_ghosts(int type, int ndim, int dims[], int width[], char *array_name, int chunk[])**
  - **For arrays with irregular distribution:**
    - **n-d Fortran** — **logical function nga_create_ghosts_irreg(type, dims, width, array_name, map, block, g_a)**
    - **C** — **int NGA_Create_ghosts_irreg(int type, int ndim, int dims[], int width[], char *array_name, int map[], int block[])**

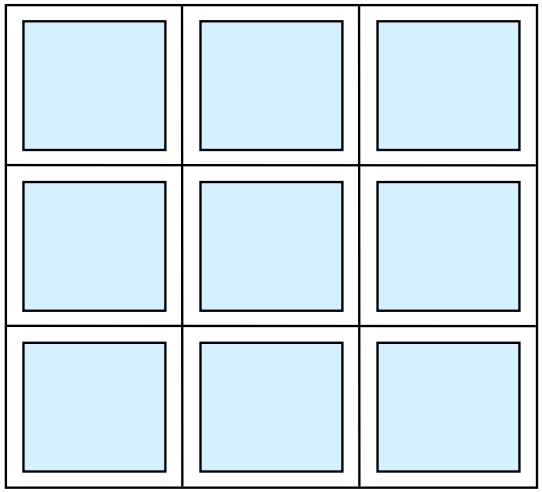
integer width(ndim) - array of ghost cell widths [input]



Ghost Cells



normal global array



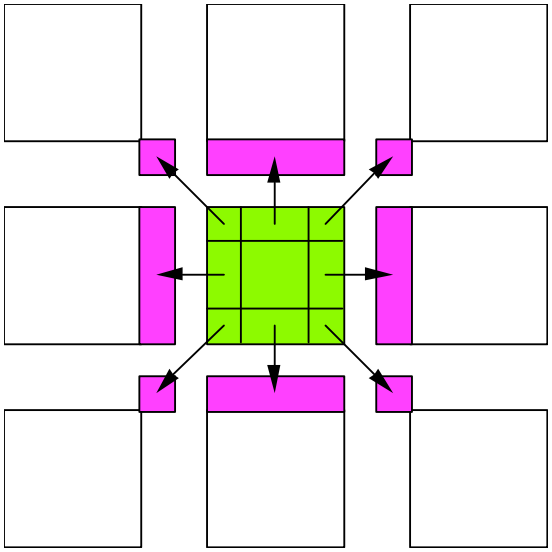
global array with ghost cells

Operations:

NGA_Create_ghosts - creates array with ghosts cells
GA_Update_ghosts - updates with data from adjacent processors
NGA_Access_ghosts - provides access to "local" ghost cell elements
NGA_Nbget_ghost_dir - nonblocking call to update ghosts cells

Ghost Cell Update

Automatically update ghost cells with appropriate data from neighboring processors. A multiprotocol implementation has been used to optimize the update operation to match platform characteristics.



Linear Algebra

- **Whole Arrays:**
  - **To add to arrays:**
    - Fortran subroutine ga_add(alpha, g_a, beta, g_b, g_c)
    - C void GA_Add(void *alpha, int g_a, void *beta, int g_b, int g_c)
  - **To multiply arrays:**
    - Fortran subroutine ga_dgemm(transa, transb, m, n, k, alpha, g_a, g_b, beta, g_c)
    - C void GA_Dgemm(char ta, char tb, int m, int n, int k, double alpha, int g_a, int g_b, double beta, int g_c)

double precision/complex/integer	alpha, beta		[input]
integer	g_a, g_b, g_c	- array handles	[input]
double/complex/int	*alpha	- scale factor	[input]
double/complex/int	*beta	- scale factor	[input]
character*1	transa, transb		[input]
integer	m, n, k		[input]
double precision	alpha, beta		[input] (DGEMM)
double complex	alpha, beta		[input] (ZGEMM)
integer	g_a, g_b		[input]
integer	g_c		[output]

Linear Algebra (cont.)

- **Whole Arrays (cont.):**
  - **To compute the element-wise dot product of two arrays:**
    - **Three separate functions for data types**
      - **Integer**
        - **Fortran ga_idot(g_a, g_b)**
        - **C GA_Idot(int g_a, int g_b)**
      - **Double precision**
        - **Fortran ga_ddot(g_a, g_b)**
        - **C GA_Ddot(int g_a, int g_b)**
      - **Double complex**
        - **Fortran ga_zdot(g_a, g_b)**
        - **C GA_Zdot(int g_a, int g_b)**

integer	**[input]**	**g_a, g_b**
integer		**GA_Idot(int g_a, int g_b)**
long		**GA_Ldot(int g_a, int g_b)**
float		**GA_Fdot(int g_a, int g_b)**
double		**GA_Ddot(int g_a, int g_b)**
DoubleComplex		**GA_Zdot(int g_a, int g_b)**

Linear Algebra (cont.)

- **Whole Arrays (cont.):**
  - **To symmetrize a matrix:**
    - **Fortran** **subroutine ga_symmetrize(g_a)**
    - **C** **void GA_Symmetrize(int g_a)**
  - **To transpose a matrix:**
    - **Fortran** **subroutine ga_transpose(g_a, g_b)**
    - **C** **void GA_Transpose(int g_a, int g_b)**

Linear Algebra (cont.)

- **Patches:**
  - **To add element-wise two patches and save the results into another patch:**
    - **Fortran** `subroutine nga_add_patch(alpha, g_a, alo, ahi, beta, g_b, blo, bhi, g_c, clo, chi)`
    - **C** `void NGA_Add_patch(void *alpha, int g_a, int alo[], int ahi[], void *beta, int g_b, int blo[], int bhi[], int g_c, int clo[], int chi[])`

integer	g_a, g_b, g_c		[input]
dbl prec/comp/int	alpha, beta	scale factors	[input]
integer	ailo, aihi, ajlo, ajhi	g_a patch coord	[input]
integer	bilo, bihi, bjlo, bjhi	g_b patch coord	[input]
integer	cilo, cihi, cjlo, cjhig_c patch coord		[input]

Linear Algebra (cont.)

- **Patches (cont.):**
  - **To perform matrix multiplication:**
    - **Fortran** **subroutine ga_matmul_patch(transa, transb, alpha, beta, g_a, ailo, aihi, ajlo, ajhi, g_b, bilo, bihi, bjlo, bjhi, g_c, cilo, cihi, cjlo, cjhi)**
    - **C** **void GA_Matmul_patch(char *transa, char* transb, void* alpha, void *beta, int g_a, int ailo, int aihi, int ajlo, int ajhi, int g_b, int bilo, int bihi, int bjlo, int bjhi, int g_c, int cilo, int cihi, int cjlo, int cjhi)**

integer	g_a, ailo, aihi, ajlo, ajhi	patch of g_a	[input]
integer	g_b, bilo, bihi, bjlo, bjhi	patch of g_b	[input]
integer	g_c, cilo, cihi, cjlo, cjhi	patch of g_c	[input]
dbl prec/comp	alpha, beta	scale factors	[input]
character*1	transa, transb	transpose flags	[input]

Linear Algebra (cont.)

- **Patches (cont.):**
  - **To compute the element-wise dot product of two arrays:**
    - **Three separate functions for data types**
      - **Integer**
        - **Fortran** nga_idot_patch(g_a, ta, alo, ahi, g_b, tb, blo, bhi)
        - **C** NGA_Idot_patch(int g_a, char* ta, int alo[], int ahi[], int g_b, char* tb, int blo[], int bhi[])
      - **Double precision**
        - **Fortran** nga_ddot_patch(g_a, ta, alo, ahi, g_b, tb, blo, bhi)
        - **C** NGA_Ddot_patch(int g_a, char* ta, int alo[], int ahi[], int g_b, char* tb, int blo[], int bhi[])
      - **Double complex**
        - **Fortran** nga_zdot_patch(g_a, ta, alo, ahi, g_b, tb, blo, bhi)
        - **C** NGA_Zdot_patch(int g_a, char* ta, int alo[], int ahi[], int g_b, char* tb, int blo[], int bhi[])

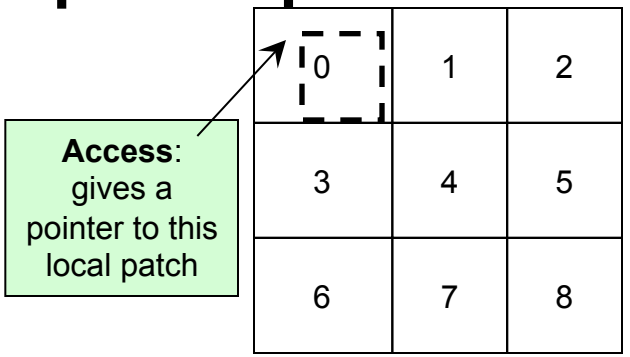
integer	g_a, g_b	[input]
integer	GA_Idot(int g_a, int g_b)	
long	GA_Ldot(int g_a, int g_b)	
float	GA_Fdot(int g_a, int g_b)	
double	GA_Ddot(int g_a, int g_b)	
DoubleComplex	GA_Zdot(int g_a, int g_b)	

Outline

- **Writing, Building, and Running GA Programs**
- **Basic Calls**
- **Intermediate Calls**
- **Writing scalable GA code with Advanced Calls**

Access (Performance from locality awareness)

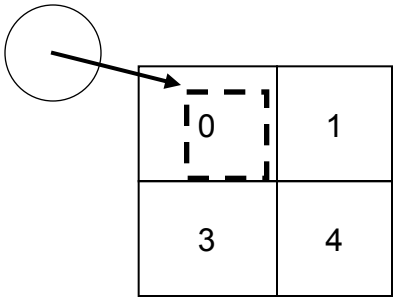
- **To provide direct access to local data in the specified patch of the array owned by the calling process:**
  - **Fortran** **subroutine nga_access(g_a, lo, hi, index, ld)**
  - **C** **void NGA_Access(int g_a, int lo[],int hi[], void *ptr, int ld[])**
  - **Processes can access the local position of the global array**
    - **Process "0" can access the specified patch of its local position of the array**
    - **Avoids memory copy**




```
call nga_create(MT_F_DBL,2,dims,'Array',chunk,g_a)
  :
call nga_distribution(g_a,me,lo,hi)
call nga_access(g_a,lo,hi,index,ld)
call do_subroutine_task(dbl_mb(index),ld(1))
call nga_release(g_a,lo,hi)
subroutine do_subroutine_task(a,ld1)
double precision a(ld1,*)
```

Locality Information (cont.)

- Global Arrays support abstraction of a distributed array object



- Object is represented by an integer handle
- A process can access its portion of the data in the global array
- To do this, the following steps need to be taken:
  - Find the distribution of an array, i.e. which part of the data the calling process owns
  - Access the data
  - Operate on the data: read/write
  - Release the access to the data

Non-blocking Operations (overlapping computation with communication)

- **The non-blocking APIs are derived from the blocking interface by adding a handle argument that identifies an instance of the non-blocking request.**
  - **Fortran**
    - **subroutine nga_nbput(g_a, lo, hi, buf, ld, nbhandle)**
    - **subroutine nga_nbget(g_a, lo, hi, buf, ld, nbhandle)**
    - **subroutine nga_nbacc(g_a, lo, hi, buf, ld, alpha, nbhandle)**
    - **subroutine nga_nbwait(nbhandle)**
  - **C**
    - **void NGA_NbPut(int g_a, int lo[], int hi[], void *buf, int ld[], ga_nbhdl_t* nbhandle)**
    - **void NGA_NbGet(int g_a, int lo[], int hi[], void *buf, int ld[], ga_nbhdl_t* nbhandle)**
    - **void NGA_NbAcc(int g_a, int lo[], int hi[], void *buf, int ld[], void *alpha, ga_nbhdl_t* nbhandle)**
    - **int NGA_NbWait(ga_nbhdl_t* nbhandle)**

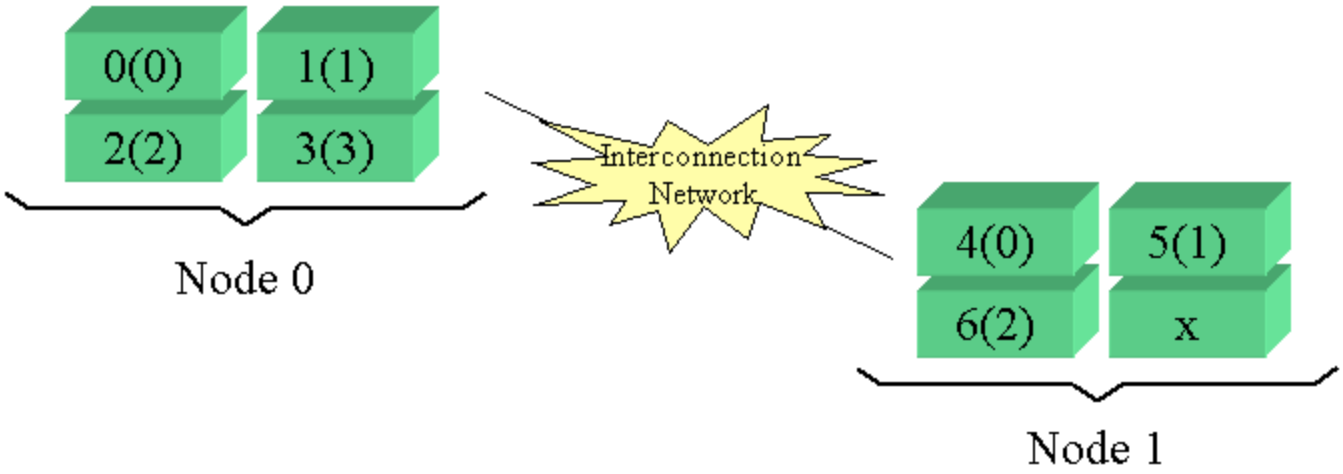
integer nbhandle - non-blocking request handle [output/input]

Non-Blocking Operations

```
double precision buf1(nmax,nmax)
double precision buf2(nmax,nmax)
   :
call nga_nbget(g_a,lo1,hi1,buf1,ld1,nb1)
ncount = 1
do while(.....)
  if (mod(ncount,2).eq.1) then
     ... Evaluate lo2, hi2
     call nga_nbget(g_a,lo2,hi2,buf2,nb2)
     call nga_wait(nb1)
     ... Do work using data in buf1
  else
     ... Evaluate lo1, hi1
     call nga_nbget(g_a,lo1,hi1,buf1,nb1)
     call nga_wait(nb2)
     ... Do work using data in buf2
  endif
  ncount = ncount + 1
end do
```

Cluster Information

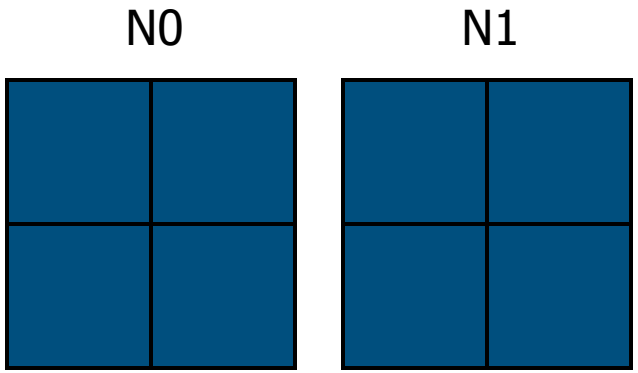
- ***Example:***
- **2 nodes with 4 processors each. Say, there are 7 processes created.**
  - **ga_cluster_nnodes returns 2**
  - **ga_cluster_nodeid returns 0 or 1**
  - **ga_cluster_nprocs(inode) returns 4 or 3**
  - **ga_cluster_procid(inode,iproc) returns a processor ID**



Cluster Information (cont.)

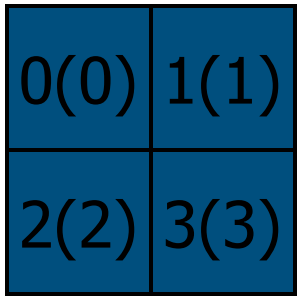
- **To return the total number of nodes that the program is running on:**
  - **Fortran integer function ga_cluster_nnodes()**
  - **C int GA_Cluster_nnodes()**

- **To return the node ID of the process:**
  - **Fortran integer function ga_cluster_nodeid()**
  - **C int GA_Cluster_nodeid()**

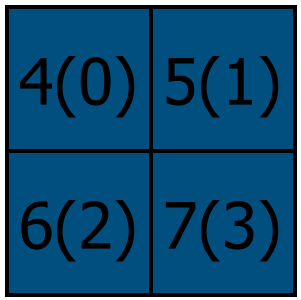


Cluster Information (cont.)

- **To return the number of processors available on node inode:**
  - **Fortran** integer function ga_cluster_nprocs(inode)
  - **C** int GA_Cluster_nprocs(int inode).



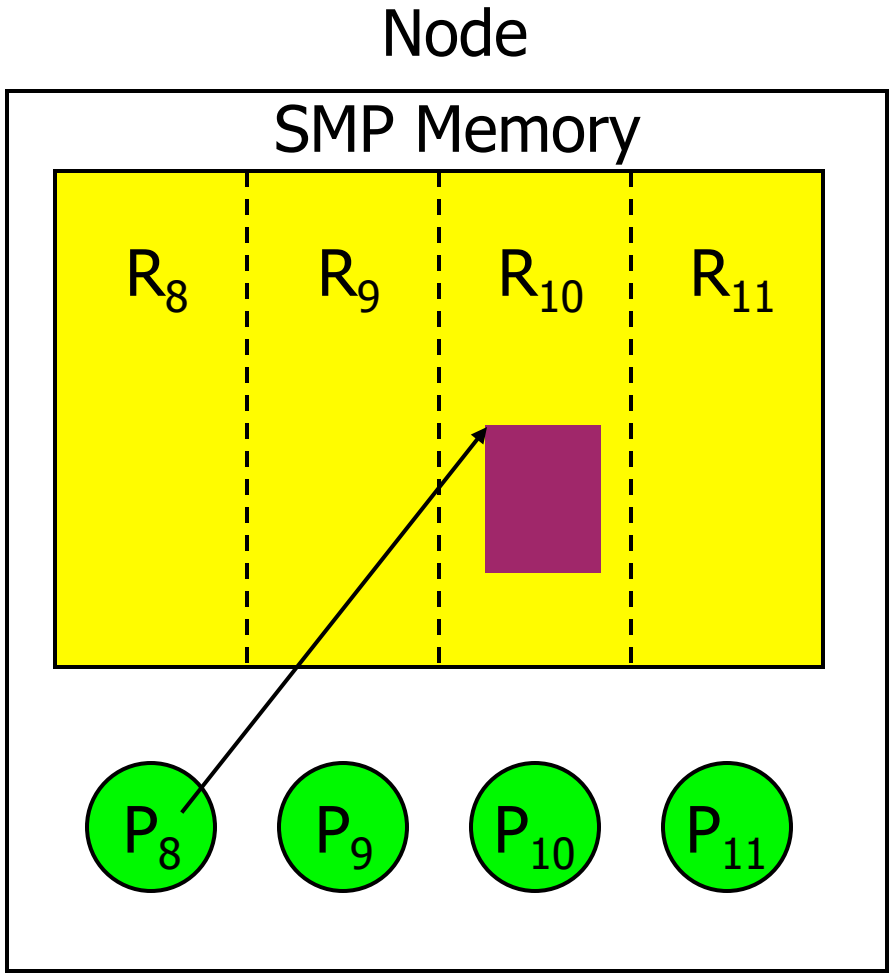
- **To return the processor ID associated with node inode and the local processor ID iproc:**
  - **Fortran** integer function ga_cluster_procid(inode, iproc)
  - **C** int GA_Cluster_procid(int inode, int iproc)



integer	inode	[input]
	inode	[input]
integer	inode,iproc	[input]
	inode,iproc	[input]

Accessing Processor Memory

ga_access



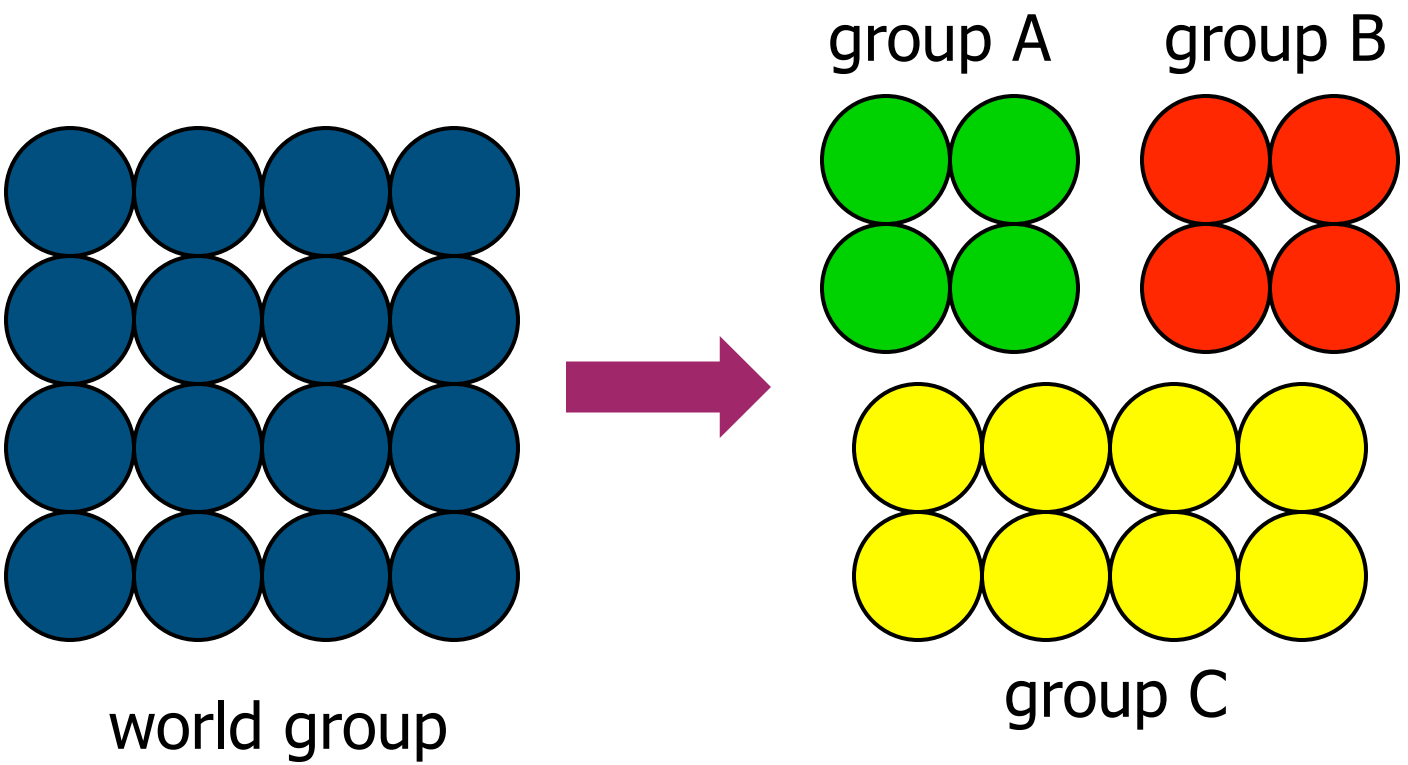
Processor Groups

- **To create a new processor group:**
  - **Fortran** integer function ga_pgroup_create(list, size)
  - **C** int GA_Pgroup_create(int *list, int size)

- **To assign a processor groups:**
  - **Fortran** logical function nga_create_config(type, ndim, dims, name, chunk, p_handle, g_a)
  - **C** int NGA_Create_config(int type, int ndim, int dims[], char *name, int p_handle, int chunk[])

integer	g_a	- global array handle	[input]
integer	p_handle	- processor group handle	[output]
integer	list(size)	-list of processor IDs in group	[input]
integer	size	-number of processors in group	[input]

Processor Groups



Processor Groups (cont.)

- To set the default processor group
  - Fortran subroutine ga_pgroup_set_default(p_handle)
  - C void GA_Pgroup_set_default(int p_handle)

- To access information about the processor group:
  - Fortran
    - integer function ga_pgroup_nnodes(p_handle)
    - integer function ga_pgroup_nodeid(p_handle)
  - C
    - int GA_Pgroup_nnodes(int p_handle)
    - int GA_Pgroup_nodeid(int p_handle)

integer p_handle - processor group handle [input]

Processor Groups (cont.)

- **To determine the handle for a standard group at any point in the program:**
  - **Fortran**
    - **integer function ga_pgroup_get_default()**
    - **integer function ga_pgroup_get_mirror()**
    - **integer function ga_pgroup_get_world()**
  - **C**
    - **int GA_Pgroup_get_default()**
    - **int GA_Pgroup_get_mirror()**
    - **int GA_Pgroup_get_world()**

```
c  create subgroup p_a
   p_a=ga_pgroup_create(list, nproc)
   call ga_pgroup_set_default(p_a)
   call parallel_task()
   call ga_pgroup_set_default(ga_pgroup_get_world()
```

```
subroutine parallel_task()
          p_b=ga_pgroup_create(new_list, new_nproc)
          call g p_b=ga_pgroup_create(new_list, new_nproc)
          call ga_pgroup_set_default(p_b)
          call parallel_subtask()
```

Lock and Mutex

- ***Lock*** **works together with** ***mutex*****.**
- **Simple synchronization mechanism to protect a critical section**
- **To enter a critical section, typically, one needs to:**
  - **Create mutexes**
  - **Lock on a mutex**
  - **Do the exclusive operation in the critical section**
  - **Unlock the mutex**
  - **Destroy mutexes**
- **The** ***create mutex*** **functions are:**
  - **Fortran logical function ga_create_mutexes(number)**
  - **C int GA_Create_mutexes(int number)**

number - number of mutexes in mutex array [input]

Lock and Mutex (cont.)

- The *destroy mutex* functions are:
  - Fortranlogical function ga_destroy_mutexes()
  - C int GA_Destroy_mutexes()

- The *lock* and *unlock* functions are:
  - Fortran
    - subroutine ga_lock(int mutex)
    - subroutine ga_unlock(int mutex)
  - C
    - void GA_lock(int mutex)
    - void GA_unlock(int mutex)

integer mutex [input] ! mutex id

Fence

- ***Fence*** **blocks the calling process until all the data transfers corresponding to the Global Array operations initiated by this process complete**

- **For example, since ga_put might return before the data reaches final destination, ga_init_fence and ga_fence allow process to wait until the data transfer is fully completed**
  - **ga_init_fence();**
  - **ga_put(g_a, ...);**
  - **ga_fence();**

- **The *initialize fence* functions are:**
  - **Fortran subroutine ga_init_fence()**
  - **C void GA_Init_fence()**

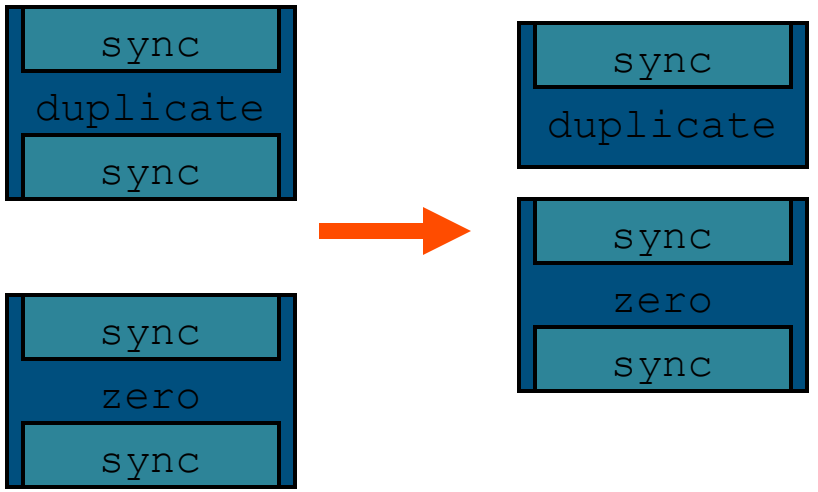
- **The *fence* functions are:**
  - **Fortran subroutine ga_fence()**
  - **C void GA_Fence()**

Synchronization Control in Collective Operations

- **To eliminate redundant synchronization points:**
  - **Fortran subroutine ga_mask_sync(prior_sync_mask, post_sync_mask)**
  - **C void GA_Mask_sync(int prior_sync_mask, int post_sync_mask)**

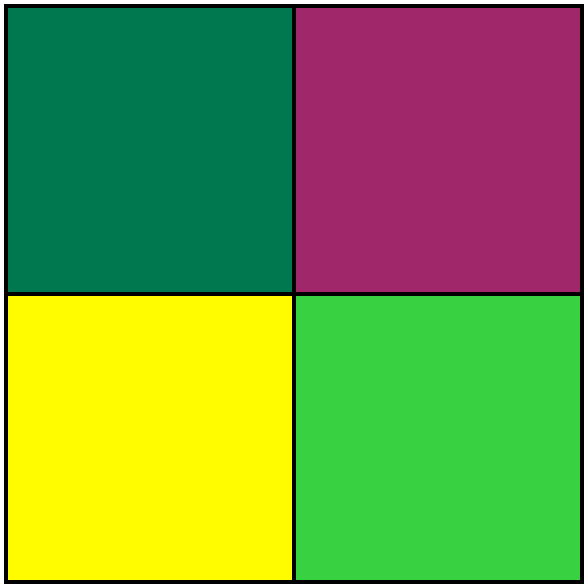
logical	**first**	**- mask (0/1) for prior internal synchronization**	**[input]**
logical	**last**	**- mask (0/1) for post internal synchronization**	**[input]**

```
call ga_duplicate(g_a, g_b)
call ga_mask(0,1)
call ga_zero(g_b)
```

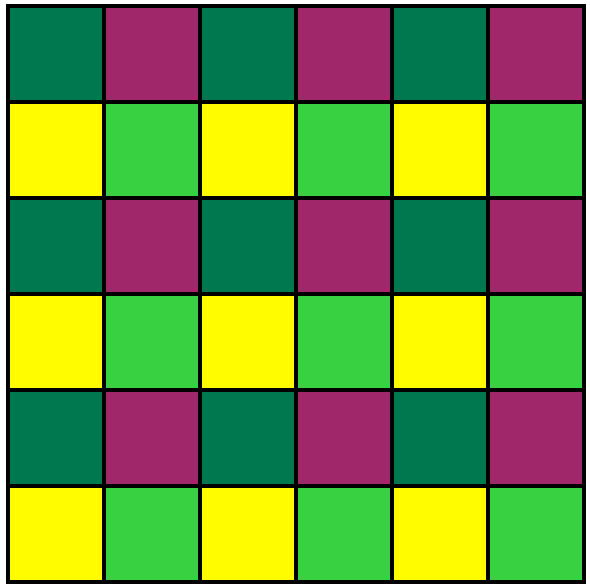


Block-Cyclic Data Distributions

Normal Data Distribution



Block-Cyclic Data Distribution



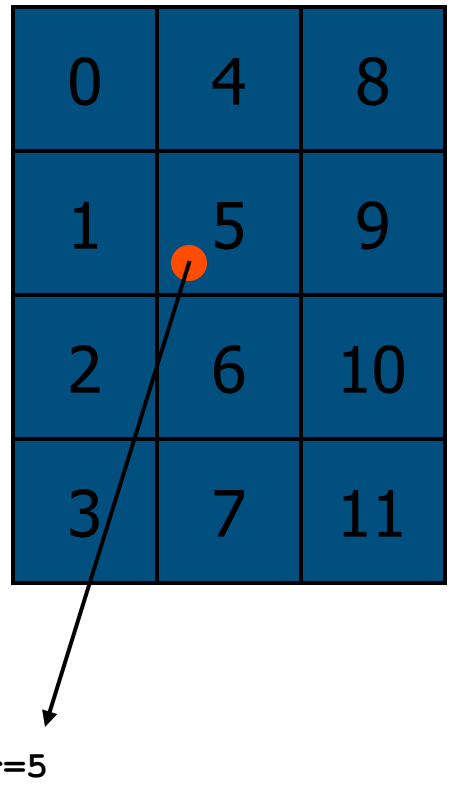
Interfaces to Third Party Software Packages

- **Scalapack**
  - **Solve a system of linear equations**
  - **Compute the inverse of a double precision matrix**
- **TAO**
  - **General optimization problems**
- **Interoperability with Others**
  - **PETSc**
  - **CUMULVS**

Locality Information

- **To determine the process ID that owns the element defined by the array subscripts:**
  - **n-Dfortran logical function nga_locate(g_a, subscript, owner)**
  - **C int NGA_Locate(int g_a, int subscript[])**

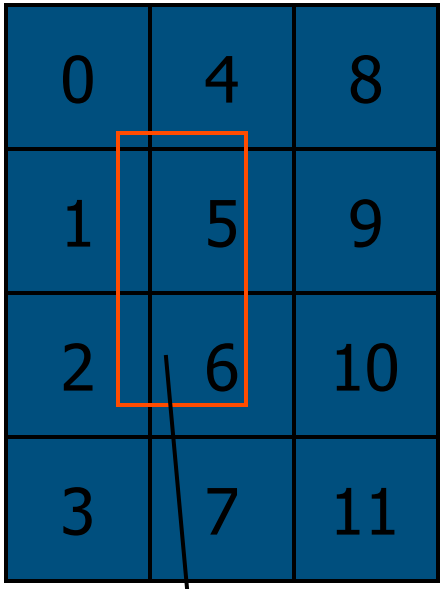
integer	**g_a**	**array handle**	**[input]**
Integer	**subscript(ndim)**	**element subscript**	**[input]**
integer	**owner**	**process id**	**[output]**



Locality Information (cont.)

- To return a list of process IDs that own the patch:
  - Fortran logical function nga_locate_region(g_a, lo, hi, map,proclist, np)
  - C int NGA_Locate_region(int g_a, int lo[], int hi[],int *map[], int procs[])

integer	np	- number of processors that own a portion of block	[output]
integer	g_a	- global array handle	[input]
integer	ndim	- number of dimensions of the global array	
integer	lo(ndim)	- array of starting indices for array section	[input]
integer	hi(ndim)	- array of ending indices for array section	[input]
integer	map(2*ndim,*)	- array with mapping information	[output]
integer	procs(np)	- list of processes that own a part of array section	[output]



procs = {0,1,2,4,5,6}

map = {lo_{01},lo_{02},hi_{01},hi_{02},
lo_{11},lo_{12},hi_{11},hi_{12},
lo_{21},lo_{22},hi_{21},hi_{22},
lo_{41},lo_{42},hi_{41},hi_{42},
lo_{51},lo_{52},hi_{51},hi_{52},
lo_{61},lo_{62},hi_{61},hi_{62}}

New Interface for Creating Arrays

- **Developed to handle the proliferating number of properties that can be assigned to Global Arrays**

Fortran

integer function ga_create_handle()

subroutine ga_set_data(g_a, dim, dims, type)

subroutine ga_set_array_name(g_a, name)

subroutine ga_set_chunk(g_a, chunk)

subroutine ga_set_irreg_distr(g_a, map, nblock)

subroutine ga_set_ghosts(g_a, width)

subroutine ga_set_block_cyclic(g_a, dims)

subroutine ga_set_block_cyclic_proc_grid(g_a, dims, proc_grid)

logical function ga_allocate(g_a)

New Interface for Creating Arrays

C

```
int GA_Create_handle()
void GA_Set_data(int g_a, int dim, int *dims, int type)
void GA_Set_array_name(int g_a, char* name)
void GA_Set_chunk(int g_a, int *chunk)
void GA_Set_irreg_distr(int g_a, int *map, int *nblock)
void GA_Set_ghosts(int g_a, int *width)
void GA_Set_block_cyclic(int g_a, int *dims)
void GA_Set_block_cyclic_proc_grid(int g_a, int
*dims, int *proc_grid)
int GA_Allocate(int g_a)
```

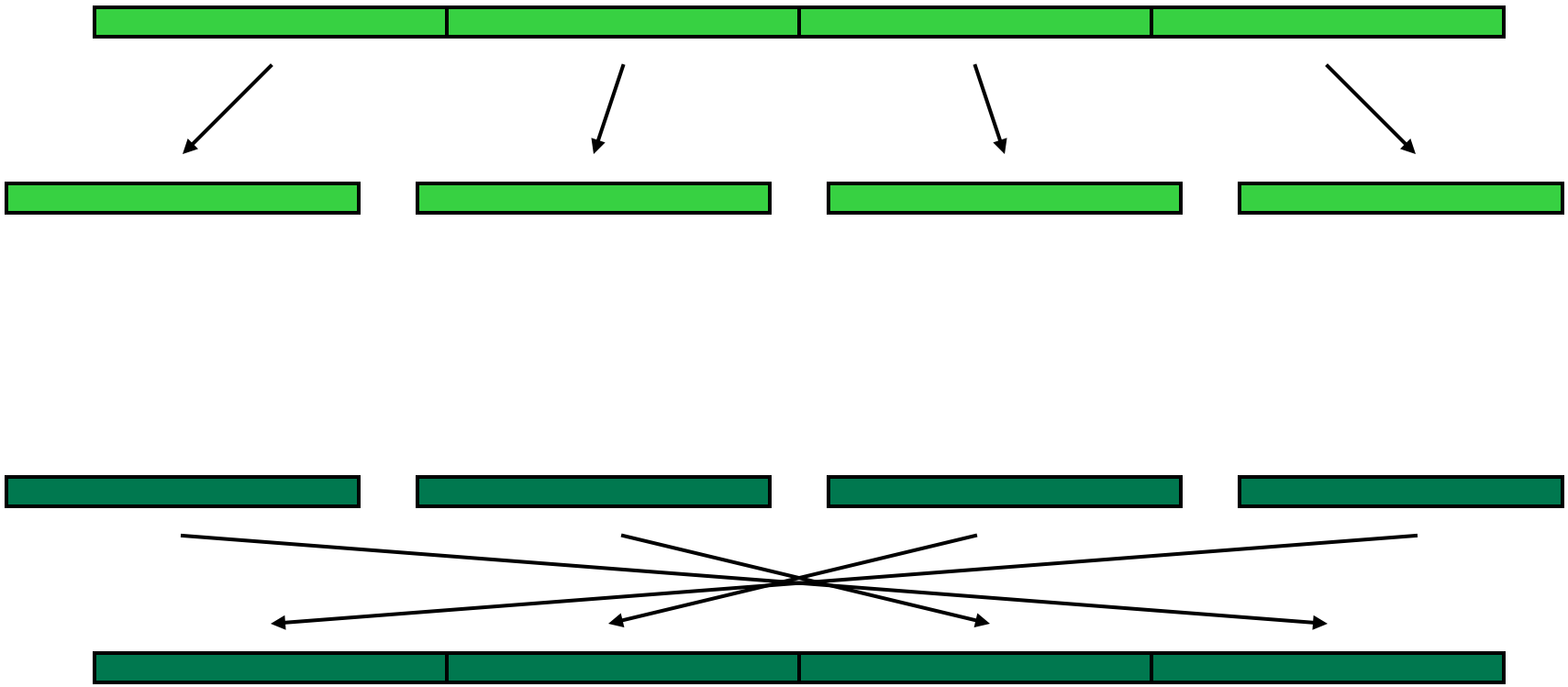
New Interface for Creating Arrays (Cont.)

```
      integer ndim,dims(2),chunk(2)
        integer g_a, g_b
        logical status
c
        ndim = 2
        dims(1) = 5000
        dims(2) = 5000
        chunk(1) = 100
        chunk(2) = 100
c
c Create global array A using old interface
c
        status = nga_create(MT_F_DBL, ndim, dims, chunk, 'array_A', g_a)
c
c Create global array B using new interface
C
        g_b = ga_create_handle()
        call ga_set_data(g_b, ndim, dims, MT_F_DBL)
        call ga_set_chunk(g_b, chunk)
        call ga_set_name(g_b, 'array_B')
        call ga_allocate(g_b)
```

Example Code

- **1-D Transpose (Fortran)**
- **1-D Transpose (C)**
- **Matrix Multiply (Fortran)**
- **Matrix Multiply (C)**

Example: 1-D Transpose



Transpose Example – C

```
int ndim, dims[1], chunk[1], ld[1], lo[1], hi[1];
int lo1[1], hi1[1], lo2[1], hi2[1];
int g_a, g_b, a[MAXPROC*TOTALELEMS],b[MAXPROC*TOTALELEMS];
int nelem, i;

/* Find local processor ID and number of processors */
int me=GA_Nodeid(), nprocs=GA_Nnodes();

/* Configure array dimensions. Force an unequal data distribution */
ndim     = 1; /* 1-d transpose */
dims[0]  = nprocs*TOTALELEMS + nprocs/2;
ld[0]    = dims[0];
chunk[0] = TOTALELEMS; /* minimum data on each process */

/* create a global array g_a and duplicate it to get g_b */
g_a = NGA_Create(C_INT, 1, dims, "array A", chunk);
if (!g_a) GA_Error("create failed: A", 0);
if (me==0) printf("  Created Array A\n");

g_b = GA_Duplicate(g_a, "array B");
if (! g_b) GA_Error("duplicate failed", 0);
if (me==0) printf("  Created Array B\n");
```

Transpose Example – C (cont.)

```
/* initialize data in g_a */
if (me==0) {
   printf("  Initializing matrix A\n");
   for(i=0; i<dims[0]; i++) a[i] = i;
   lo[0]  = 0;
   hi[0] = dims[0]-1;
   NGA_Put(g_a, lo, hi, a, ld);
}


/* Synchronize all processors to guarantee that everyone has data before proceeding
   to the next step. */
GA_Sync();


/* Start initial phase of inversion by inverting the data held locally on each processor.
   Start by finding out which data each processor owns. */
NGA_Distribution(g_a, me, lo1, hi1);


/* Get locally held data and copy it into local buffer a  */
NGA_Get(g_a, lo1, hi1, a, ld);


/* Invert data locally */
nelem = hi1[0] - lo1[0] + 1;
for (i=0; i<nelem; i++) b[i] = a[nelem-1-i];
```

Transpose Example – C (cont.)

```
/* Invert data globally by copying locally inverted blocks into
 * their inverted positions in the GA */
lo2[0] = dims[0] - hi1[0] -1;
hi2[0] = dims[0] - lo1[0] -1;
NGA_Put(g_b,lo2,hi2,b,ld);

/* Synchronize all processors to make sure inversion is complete */
GA_Sync();

/* Check to see if inversion is correct */
if(me == 0) verify(g_a, g_b);

/* Deallocate arrays */
GA_Destroy(g_a);
GA_Destroy(g_b);
```

Transpose Example - Fortran

```
      integer dims(3), chunk(3), nprocs, me, i, lo(3), hi(3), lo1(3)
      integer hi1(3), lo2(3), hi2(3), ld(3), nelem
      integer g_a, g_b, a(MAXPROC*TOTALELEMS), b(MAXPROC*TOTALELEMS)
      integer heap, stack, ichk, ierr
      logical status

      heap = 300000
      stack = 300000
c
c   Initialize communication library
c
#ifdef USE_MPI
      call mpi_init(ierr)
#else
      call pbeginf
#endif
c
c   Initialize GA library
c
      call ga_initialize()
```

Transpose Example – Fortran (cont.)

```
c
c   Find local processor ID and number of processors
c
      me = ga_nodeid()
      nprocs = ga_nnodes()
      if (me.eq.0) write(6,101) nprocs
  101 format('Using ',i4,' processors')
c
c   Allocate memory for GA library
c
      status = ma_init(MT_F_DBL, stack/nprocs, heap/nprocs)
c
c   Configure array dimensions. Force an unequal data distribution.
c
      dims(1) = nprocs*TOTALELEMS + nprocs/2
      ld(1) = MAXPROC*TOTALELEMS
      chunk(1) = TOTALELEMS     ! Minimum data on each processor
c
c   Create global array g_a and then duplicate it to get g_b
c
      status = nga_create(MT_F_INT, NDIM, dims, "Array A", chunk, g_a)
      status = ga_duplicate(g_a, g_b, "Array B")
```

Transpose Example – Fortran (cont.)

```
c
c   Initialize data in g_a
c
      do i = 1, dims(1)
        a(i) = i
      end do
      lo1(1) = 1
      hi1(1) = dims(1)
c
c   Copy data from local buffer a to global array g_a. Only do this for
c   processor 0.
c
      if (me.eq.0) call nga_put(g_a, lo1, hi1, a, ld)
c
c   Synchronize all processors to guarantee that everyone has data
c   before proceeding to the next step.
c
      call ga_sync
```

Transpose Example – Fortran (cont.)

```
c
c   Start initial phase of inversion by inverting the data held locally on
c   each processor. Start by finding out which data each processor owns.
c
      call nga_distribution(g_a, me, lo, hi)
c
c   Get locally held data and copy it into local buffer a
c
      call nga_get(g_a, lo, hi, a, ld)
c
c   Invert local data
c
      nelem = hi(1) - lo(1) + 1
      do i = 1, nelem
        b(i) = a(nelem - i + 1)
      end do
c
c   Do global inversion by copying locally inverted data blocks into
c   their inverted positions in the GA
c
      lo2(1) = dims(1) - hi(1) + 1
      hi2(1) = dims(1) - lo(1) + 1
      call nga_put(g_b, lo2, hi2, b, ld)
```

Transpose Example – Fortran (cont.)

```
c
c   Synchronize all processors to make sure inversion is complete
c
      call ga_sync()
c
c   Check to see if inversion is correct. Start by copying g_a into local
c   buffer a, and g_b into local buffer b.
c
      call nga_get(g_a, lo1, hi1, a, ld)
      call nga_get(g_b, lo1, hi1, b, ld)
      ichk = 0
      do i = 1, dims(1)
        if (a(i).ne.b(dims(1)-i+1) .and. me.eq.0) then
          write(6,111) i,a(i),b(dims(1)-i+1)
  111     format('Mismatch at ',3i8)
          ichk = ichk + 1
        endif
      end do
      if (ichk.eq.0.and.me.eq.0) write(6,*) 'Transpose OK'
```

Transpose Example – Fortran (cont.)

```
c
c   Deallocate memory for arrays and clean up GA library
c
      if (me.eq.0) write(6,*) 'Terminating...'
      status = ga_destroy(g_a)
      status = ga_destroy(g_b)
      call ga_terminate
#ifdef USE_MPI
      call mpi_finalize
#else
      call pend
#endif
      stop
      end
```

Matrix Multiply Example – C

```
int dims[NDIM], chunk[NDIM], ld[NDIM];
int lo[NDIM], hi[NDIM], lo1[NDIM], hi1[NDIM];
int lo2[NDIM], hi2[NDIM], lo3[NDIM], hi3[NDIM];
int g_a, g_b, g_c, i, j, k, l;

/* Find local processor ID and the number of processors */
int me=GA_Nodeid(), nprocs=GA_Nnodes();

/* Configure array dimensions. Force an unequal data distribution */
for(i=0; i<NDIM; i++) {
   dims[i]  = TOTALELEMS;
   ld[i]= dims[i];
   chunk[i] = TOTALELEMS/nprocs-1; /*minimum block size on each process*/
}
```

Matrix Multiply Example – C (cont.)

```
/* create a global array g_a and duplicate it to get g_b and g_c*/
g_a = NGA_Create(C_DBL, NDIM, dims, "array A", chunk);
if (!g_a) GA_Error("create failed: A", NDIM);
if (me==0) printf("  Created Array A\n");

g_b = GA_Duplicate(g_a, "array B");
g_c = GA_Duplicate(g_a, "array C");
if (!g_b || !g_c) GA_Error("duplicate failed",NDIM);
if (me==0) printf("  Created Arrays B and C\n");

/* initialize data in matrices a and b */
if(me==0)printf("  Initializing matrix A and B\n");
k = 0; l = 7;
for(i=0; i<dims[0]; i++) {
   for(j=0; j<dims[1]; j++) {
  a[i][j] = (double)(++k%29);
  b[i][j] = (double)(++l%37);
   }
}
```

Matrix Multiply Example – C (cont.)

```
/*  Copy data to global arrays g_a and g_b */
lo1[0] = 0;
lo1[1] = 0;
hi1[0] = dims[0]-1;
hi1[1] = dims[1]-1;
if (me==0) {
   NGA_Put(g_a, lo1, hi1, a, ld);
   NGA_Put(g_b, lo1, hi1, b, ld);
}

/*  Synchronize all processors to make sure everyone has data */
GA_Sync();

/* Determine which block of data is locally owned. Note that
   the same block is locally owned for all GAs. */
NGA_Distribution(g_c, me, lo, hi);
```

Matrix Multiply Example – C (cont.)

```
/* Get the blocks from g_a and g_b needed to compute this block in
   g_c and copy them into the local buffers a and b. */
lo2[0] = lo[0];
lo2[1] = 0;
hi2[0] = hi[0];
hi2[1] = dims[0]-1;
NGA_Get(g_a, lo2, hi2, a, ld);

lo3[0] = 0;
lo3[1] = lo[1];
hi3[0] = dims[1]-1;
hi3[1] = hi[1];
NGA_Get(g_b, lo3, hi3, b, ld);

/* Do local matrix multiplication and store the result in local
   buffer c. Start by evaluating the transpose of b. */
for(i=0; i < hi3[0]-lo3[0]+1; i++)
   for(j=0; j < hi3[1]-lo3[1]+1; j++)
  btrns[j][i] = b[i][j];
```

Matrix Multiply Example – C (cont.)

```
/* Multiply a and b to get c */
for(i=0; i < hi[0] - lo[0] + 1; i++) {
   for(j=0; j < hi[1] - lo[1] + 1; j++) {
  c[i][j] = 0.0;
  for(k=0; k<dims[0]; k++)
 c[i][j] = c[i][j] + a[i][k]*btrns[j][k];
   }
}

/* Copy c back to g_c */
NGA_Put(g_c, lo, hi, c, ld);

verify(g_a, g_b, g_c, lo1, hi1, ld);

/* Deallocate arrays */
GA_Destroy(g_a);
GA_Destroy(g_b);
GA_Destroy(g_c);
```

Sources of Information

- http://www.emsl.pnl.gov/docs/global/